Cambria: Schema Evolution in Distributed Systems with Edit Lenses

Geoffrey Litt
MIT CSAIL
Cambridge, MA, USA
glitt@mit.edu

Peter van Hardenberg
Ink & Switch
San Francisco, CA, USA
pvh@inkandswitch.com

Orion Henry
Ink & Switch
Miami, FL, USA
orion@inkandswitch.com

Abstract

Schema evolution is a pervasive challenge in distributed systems. Developers often resort to ad hoc solutions that mix compatibility concerns with application code. We propose a principled replacement: an isolated software layer that uses bidirectional lenses to translate data between schemas on demand to maintain backward and forward compatibility.

We have implemented these ideas in a TypeScript library called Cambria, and integrated it with the Automerge CRDT to support peer-to-peer editing. We share reflections from building a decentralized issue tracker application and successfully using Cambria to maintain compatibility across versions. We also present our vision for a “Cambrian Era” for software, where better schema evolution tools enable people to customize their software more freely while preserving the ability to collaborate.

CCS Concepts: • Information systems → Mediators and data integration; • Human-centered computing → Collaborative and social computing systems and tools; • Software and its engineering → Peer-to-peer architectures.

Keywords: schema evolution, bidirectional lenses, CRDTs

ACM Reference Format:

Geoffrey Litt, Peter van Hardenberg, and Orion Henry. 2021. Cambria: Schema Evolution in Distributed Systems with Edit Lenses. In *8th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC’21)*, April 26, 2021, Online, United Kingdom. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3447865.3457963>

1 Introduction

As distributed systems evolve, they often need to update data schemas while maintaining backward and forward compatibility. Consider the following Javascript code, which loads a task from a document store and adds a tag:

```
let task = fetchObjectFromDatabase()
if (task.tags && Array.isArray(task.tags)) {
  task.tags.push(newTag);
} else {
  // Handle old tasks without a tags array
  task.tags = [newTag];
}
```

The core behavior is simple. But the code is made more complicated by considering the case where the document was created by an older version of the code, where the tags property did not exist yet. This scenario is an example of *schema evolution*: updating a schema while maintaining compatibility with older data [14]. Schema evolution issues have been studied in databases [1, 17], object stores [12, 15], and serialization formats [7].

Schema evolution is particularly challenging in decentralized systems. In a centralized architecture, a database can define a single current schema, and perform one-time upgrades from one schema to the next. But in a decentralized architecture, there might be many peer devices running different versions of the software, each with their own replicas of some shared application state. It is not necessarily possible to agree on a single schema across all devices.

In order to write code that works correctly with data created in older or newer versions of a schema, developers frequently resort to ad hoc solutions like the one shown above. The code snippet exemplifies the Shotgun Parsing anti-pattern [11], where data transformation and validation is spread throughout core application logic, making the code more error-prone and difficult to understand.

We propose a more principled solution: an isolated layer called Cambria that translates data between schemas on demand. Developers specify translation logic with *bidirectional lenses* [4–6], programs that simultaneously specify both directions of a data transformation. They can then write application code that only considers a single data schema, without any compatibility logic. Cambria translates data between schemas, ensuring that the code is forward and backward compatible. We have also integrated Cambria with the Automerge JSON CRDT [8] to support convergent decentralized editing across different JSON schemas.

To motivate the key ideas, we start with a case study of using Cambria to build a decentralized issue tracker with

This work is licensed under a Creative Commons Attribution International 4.0 License.

PaPoC’21, April 26, 2021, Online, United Kingdom

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8338-7/21/04.

<https://doi.org/10.1145/3447865.3457963>

Figure 1. Two versions of the issue tracker application, showing the same document through different schemas.

realtime collaboration. We show how we were able to ergonomically maintain compatibility across versions with different data schemas (Section 2). We then describe the design of the system (Section 3), which includes a bidirectional lens language, a graph of schemas connected by lenses, and a structure for persisting documents in a JSON CRDT that can be simultaneously edited through multiple schemas. While we build on existing literature in bidirectional lenses, this system design in combination provides a novel approach to managing schema compatibility in CRDTs. Next, we share qualitative reflections on the challenges that emerged in the case study and discuss their implications for cross-schema collaboration in distributed systems (Section 4).

Finally, we conclude by presenting our vision for a “Cambrian” software ecosystem. Schema mapping tools aren’t just useful for achieving compatibility across versions of the same software; they can also be used to achieve compatibility between completely different pieces of software. We describe how schema mapping could give users more agency for choosing and customizing their software, while still retaining the ability to collaborate together on shared data.

2 Case Study

Before getting to the technical details of Cambria, in this section we share an example scenario of using the tool to maintain compatibility in a decentralized application.

Our group has been developing local-first [9] peer-to-peer collaboration software using the PushPin architecture [16], built on the Automerge CRDT library [8]. In this architecture, each peer runs a desktop application storing its own local copy of the data, and edits are exchanged between peers when network access is available. A major challenge of this architecture has been handling the case where users are running different versions of the application while collaborating on the same data.

To ensure our library could solve these real needs, we built a sample application—a software issue tracker—in the local-first architecture, and co-developed the Cambria library to solve compatibility problems that emerged along the way. Our goal was to achieve full compatibility: users should be able to collaborate on a document regardless of the software version they are running. To force ourselves to manage the kinds of schema changes that emerge in real software projects, we started using the issue tracker ourselves early on, and added features to the application as we needed them.

Whenever we needed to update the data schema, we wrote a lens to express the transformation between the old schema and the new one. Then, we would write application code that assumed that any persistent documents were stored in a single “current” schema, without any compatibility concerns. If the application opened a document written in an older or newer version of the code, Cambria would internally convert the document to the expected schema, so the code would never break. Cambria also outputs static type definitions, which made it easier for us to write correct code that worked with the current schema definition.

We were generally able to maintain compatibility across over a dozen schema versions, with some exceptions.¹ Different members of the team could run different versions of the software while editing the same list of tasks.

As we developed the application further, we added lens operators to Cambria to meet our schema transformation needs. Of course, while we were able to effectively use our system for this one particular application, we don’t claim that our system can handle the transformations needed by every application.

Below, we examine some changes we made to our data schema for tasks in our issue tracker. These changes were applied on top of a simple initial schema for tasks, shown below as a TypeScript type definition:

```
type Task = {
  name: string;
  description: string;
  complete: boolean;
  assignee: string;
}
```

¹One problem we encountered was that we were changing the underlying storage format used by Cambria itself; this kind of problem seems unlikely to affect a project that was merely using Cambria and not developing it.

2.1 Adding an Array Property

We decided to add tags to each task, represented by an array of strings. As shown in Section 1, even such a simple change can require writing messy compatibility code.

Using Cambria, we can write a short lens program that specifies the new property to add to the schema, shown below in a YAML-based syntax:

```
- add:
  name: tags
  type: array
```

When our code loads a task created by an older version, the document is translated through this lens so that it conforms to the new schema. Newly added properties are populated with default values. The automatic default for an array property is [], so our tags property is filled in with an empty array. The application code now safely assume that there is an array present.

2.2 Converting Data Values

Initially, our application tracked the status of each task with a boolean complete field, but we later decided to replace that with a status field that could take on three values: todo, in progress, and done. We wrote a lens that expressed this conversion:

```
- rename:
  source: complete
  destination: status
- convert:
  name: status
  sourceType: boolean
  destinationType: string
  mapping:
    - 'false': todo
    - 'true': done
    - todo: false
    - inProgress: false
    - done: true
    - default: false
```

First, the lens renames the complete property to status. Then, it converts the value of the property from a boolean to a corresponding string. For example, complete: false is mapped to status: todo.

When the user on the old version checks a task to mark it complete, the new version updates the status of the task to the corresponding string value, done (Figure 1). Edits also work in the other direction: if the user on the new version sets the status back to todo, the old version shows an unchecked checkbox. Each client edits the task using its native data schema, with no knowledge of the other schema, and Cambria translates the edits back and forth. In many systems renaming a field is a complicated process, but here it just requires writing a small lens.

This lens is a lossy conversion, since the boolean checkbox on the old client cannot reflect whether an incomplete task is in progress or not. Nevertheless, this lossy conversion might be adequate to allow two users to collaborate across versions. If at some point the costs outweigh the benefits, the best option might be to require all collaborators to upgrade to the same version. Cambria doesn't force users to collaborate across versions with imperfect compatibility; it simply provides the option of doing so.

2.3 Mapping a Scalar to an Array

Our issue tracker initially supported assigning one person to each task. We later decided we wanted the ability to assign multiple people to a task. We wrote a lens which renames the field from assignee to assignees, and then uses the wrap operator, which translates a scalar value into an array:

```
- rename:
  source: assignee
  destination: assignees
- wrap: { name: assignees }
```

The wrap operator works as follows. When the array contains a single element, both sides show the same information. When multiple elements are present in the array, the scalar value reflects the first element in the array. When the array is empty, the scalar is set to null. This operator presents some difficult design tradeoffs; see Section 4.0.1 for more details.

3 System Design

Cambria is implemented as a Typescript library, and the source code is publicly available². The system has three main components:

- 1) **A lens language** that specifies bidirectional data transformations between a pair of schemas. We also generate static type signatures and schema definitions from lenses, to ensure that data payloads and type signatures maintain aligned.
- 2) **A lens graph** that provides a mechanism for composing multiple lenses and schemas. The graph structure offers the flexibility to connect schemas in useful ways beyond a simple list.
- 3) **A version-tagged CRDT**: a structure for persisting documents in a JSON CRDT that can be read from and written to simultaneously by nodes using different schemas.

The lens language itself is directly based on prior work by Hoffmann et al. [6]. However, to our knowledge, our work is the first to apply lenses to decentralized editing, by composing them into a graph structure and integrating them with a CRDT.

²<https://github.com/inkandswitch/cambria>

Also, this separation of responsibilities makes Cambria useful in other contexts beyond our case study architecture. For example, we prototyped a library for translating web API payloads, which does not use the version-tagged edit log since it doesn't manage persisted documents.

3.1 Lens Language

At the heart of Cambria is a domain-specific language for transforming JSON data between related schemas. We chose to build a language that translates edit operations rather than translating entire documents. This naturally aligned with the architecture of our sample application, where peers exchange operations rather than exchanging entire new documents. (Of course, the language can still translate an entire document, simply by representing it as a series of operations.)

We created a language directly based on Hofmann et al.'s Edit Lenses [6]. Here we present a simplified summary of the main ideas, but refer the reader to their paper for an in-depth explanation. An edit lens defines a pair of functions $\Rightarrow$ and $\Leftarrow$ which can translate edits back and forth between two related data structures. Consider two compound data structures x and y . In our case we operate on JSON objects, which contain nested maps and lists. We can define a consistency relation K that should hold between the two structures: $(x, y) \in K$.

Consider an arbitrary valid edit dx on x , which produces an updated structure x' . The $\Rightarrow$ function translates dx into a corresponding edit dy , which applied to y produces y' . For the edit lens to be well-formed, the transformation must satisfy two properties:

- dy must be a valid edit that can be applied to y
- The consistency relation is preserved: $(x', y') \in K$

Similar properties must hold in the opposite direction for the $\Leftarrow$ function, which translates edits on y into edits on x .

Some lenses and their corresponding consistency relations are straightforward. For example, consider the case where x and y have the same schema, except that y has an additional field `foo`. There is a trivial relation that makes sense here: y can contain any value for `foo`, and x must contain no value for `foo`. The edit lens that preserves this relationship is simple: any dy modifying `foo` generates a no-op on x ; all other edits are passed through unchanged in both directions. The effect is that the two data structures receive the same edits, except that x is unaware of any edits to `foo`.

Other lenses are more nuanced. Consider the case where x is an array, and y is a scalar, as shown in Section 2.3. A reasonable consistency relation is to say that the first element of x must always be y . However, this does not fully specify the behavior. For example, what happens when x is an empty array? Even if we further specify the consistency relation, there are often multiple possible edit lenses that would satisfy the rules, which might align more or less closely to user expectations. See Section 4 for more discussion of this point.

Our language contains the following primitive lens operators, grouped into related pairs with their inverses:

- **add / remove**: Add or remove a property
- **rename**: Rename a property
- **hoist / plunge**: Move a property up or down in the tree of nested maps
- **wrap / head**: Turn a scalar into an array, and vice versa
- **convert**: Transform a value based on a predefined hardcoding mapping.

A lens program can be constructed by simply concatenating a sequence of primitive operations, as shown in the examples in Section 2. The language also contains two higher-order operators which take a lens as an argument:

- **in**: Run a given lens inside a sub-object within some property
- **map**: Run a given lens for each element of an array

Many other lens operators could be added while fitting into this general framework. For example, any invertible math operation could be provided as a lens operator. However, our case study showed that this minimal subset was sufficient to perform many useful transformations in a realistic context.

3.1.1 Generating schema definitions from lenses. In order to eliminate the need for ad-hoc validations, it is critical that programmers can understand the schema of their data at compile time, and trust that the data their program receives actually matches that schema at runtime. In order to provide such guarantees, Cambria creates two additional artifacts based on the same lens definitions used for runtime data transformations: 1) static type definitions in Typescript, and 2) runtime-checkable schema definitions in JSON Schema (a schema description language for JSON, represented itself as JSON). The Typescript types provide the programmer with a clear view of the data schema while programming. The JSON Schema is used to validate data received from peers; if a peer sends data that claims to be in some schema but is not actually valid, the program can reject that data early on before continuing on with further execution.

To implement this feature, we developed an additional context for interpreting lens programs, where the inputs/outputs are *schema definitions*, rather than runtime data. Every lens operator has a natural analog in the domain of static schemas. For example, consider the `add` lens. Given some schema s , if we apply `add`, we obtain a new schema t that has that new property added.

This interpreter context runs lens programs on JSON Schema files, and an automated transformation from JSON Schema to TypeScript produces static type definitions. In our usage of Cambria, we never manually defined TypeScript types or JSON Schema definitions for our application. Rather, we wrote lenses (starting from the empty schema) to define

Figure 2. Schemas are connected by lenses. An edit is translated between schemas by traversing the shortest path.

both our schemas and our transformation logic, which ensured that the two always matched.

3.2 Lens Graph

So far, we have discussed lenses between a pair of schemas. But what happens when our system needs to accommodate more than two schemas?

One option would be to define an ordered list of schemas connected by lenses. To translate an edit between two schemas in the list, the edit would be sequentially passed through all the lenses connecting the two schemas. The resulting transformation would be valid since the concatenation of two lenses is always a lens. This approach might work for a centralized system where schemas progress in a single list of versions, but we wanted to support other use cases:

- Decentralized systems, where no single authority can decide on the next “version” of a schema.
- Branching and merging. For example, two developers could each write some test data on their development branches, and then merge the test data when they merge the code branches together.
- Schema *mapping*, in addition to schema *evolution*. Rather than just connecting versions of a schema within an application, we also wanted to support connecting schemas across different applications.

To support these use cases, we used a more general structure to store schemas: a graph where schemas are nodes and lenses are edges. The graph is initialized with a special node representing the empty schema. To add a new schema to the graph, the user can define a lens from an existing schema in the graph, generating a new node and edge.

Because lenses are bidirectional, every edge of the graph can be traversed in either direction. However, the edges do have some meaningful direction, because a lens operates differently in the forward and backward direction.

To translate an edit operation to a different schema in the graph, we find the shortest path and pass the edit through all the lenses on that path (Figure 2). As before, because the concatenation of lenses is a lens, this produces a valid lens between the two schemas.

3.3 Version-tagged CRDT

In most databases, a document conforms to a single schema at a given time. Database migrations can perform a one-off conversion from data from one version to another, but it’s typically not possible to simultaneously accommodate reads and writes of the same data in terms of different schemas. Such an approach was not sufficient for our decentralized issue tracker, since we wanted to support realtime collaboration on a document between peers running different versions of the software.³

Another requirement of our system was to support convergent decentralized collaboration, where peers could concurrently edit a document and eventually arrive at a consistent state after exchanging all of their edits.

Both of these requirements naturally aligned with storing the document as a log of edit operations. To leverage an existing CRDT implementation for convergent editing, we integrated Cambria with the Automerge CRDT [8]. The main data structure in Automerge is a log of all changes that have been made to a document (e.g., setting keys on an object, inserting values into lists, and creating new objects). Our modified version of Automerge maintains this operation log, but additionally tags operations with schema metadata to facilitate lens translations.

As shown in Figure 3, when a client makes a change, the new operation in the log is tagged with the writer schema—the schema that was used to make the change. When a client reads the document by replaying the log of changes, each change is first translated by Cambria into the desired schema before being applied. The result is a log of patches in terms of the reader schema, which can be replayed into a final document in that schema.

We achieve convergent editing by building on the behavior of the Automerge CRDT. For example, consider a document represented in two schemas, S_1 and S_2 . S_1 contains an array property todos; the lens from S_1 to S_2 renames the property to tasks. Concurrently, an application using S_1 inserts some objects into the todos array, and an application using S_2 inserts objects into its tasks array. In this case, Cambria’s lenses map both writes to the same array, even though each writer used a different property name. Then, Automerge handles deterministically ordering the list insertions so the array contents converge for both replicas. The result is that S_1

³Consider the case of a document represented in two schemas, S_1 and S_2 . S_1 contains the properties foo and bar. The lens from S_1 to S_2 removes the field bar and adds a new field baz. We would like the document to be able to contain data for foo, bar, and baz, which is impossible if it must take on the schema of only S_1 or S_2 .

Figure 3. Cambria transforms each write operation from the original writer schema to the desired reader schema.

and S_2 see the same list, under the todos and tasks property respectively.

In general, the lens and CRDT behavior are orthogonal and integrate without any particular challenges. One interesting exception is the head and wrap lenses, which convert between a scalar and a list. Because the Automerge CRDT handles those two kinds of values differently, the interaction between two documents is subtle. We discuss the details later in Section 6.0.1.

3.3.1 Storing lenses. Another goal of our system was to create self-describing documents. In order to maximize forward compatibility, we wanted an old version of the software to be able to open documents created by new versions, without requiring any code updates. This meant that any documents written by the new version would need to contain enough information for older versions to correctly read and write changes.

Our solution was to store a serialized version of the lens graph in the document itself. When a client writes to a document using a schema not already present in the lens graph, it writes a special change to the edit log which records the source code for the new lens (and, because schemas are derived from lenses, indirectly registers the new schema). As long as the lens language itself is stable, this means that other peers can still interpret the new changes in their own schema, even if running an older version of the code.

This approach of storing a lens graph together with the document does not make sense in all situations. For example, for an application with many documents using shared schemas, there would be wasted space from duplication of the graph inside each document. In such a context, it would make more sense to store a single lens graph in some shared location across multiple documents.

4 Reflections

As described in Section 2, we were able to successfully use Cambria to evolve the schema for a decentralized issue tracker over several months of real usage. Our application code only considered the latest schema, without compatibility logic. We achieved compatibility in both directions: old clients

could open documents created by new clients, and vice versa. Furthermore, different versions could simultaneously edit the same list of issues. We believe this provides preliminary evidence that this approach may be a viable and improved development paradigm for schema evolution in decentralized applications.

Below we share several qualitative observations from our experience using lenses for schema evolution.

4.0.1 Interoperability requires trading off between irreconcilable design goals. Today, we generally tend to view compatibility as a binary: two systems are either technically compatible, or not. But with a flexible schema evolution system, compatibility becomes more of a human-centered design question: how far can two pieces of software diverge before it becomes impossible to usefully collaborate between them? Lens evolutions cannot magically make two incompatible pieces of software work perfectly together; the goal is merely to enable developers to maximize compatibility within realistic limits.

We found that when two applications use sufficiently different data representations, any interoperability design requires making tradeoffs between several desirable design goals:

- **consistency:** both sides see a meaningfully equivalent view of the world
- **conservation:** neither side operates on data they can't observe
- **predictability:** the local intent of every operation is preserved

Consider once again the example of two versions of our issue tracker: one with a single assignee per task, another with multiple assignees. When there are multiple people assigned to a task, the older app version only shows the first assignee. Let's say a user on the older version removes that single assignee. What should happen in the multiple-assignee version?

First, we'll demand consistency as non-negotiable. If both sides don't see some meaningfully related view of the same reality, we're not really working on the same data.

We could interpret the “delete” operation as requesting deletion of all the elements in the array, but now we’ve invalidated our **conservation** goal. The old version of the program has unassigned other people from the task without knowing they were assigned in the first place.

Another option is to just remove the first element of the array. But now, consistency demands that the old program should change the single assignee to a different value: the new first assignee. For a user of the old program, it’s surprising to remove an assignee and see a different name appear in the box; this option violates the **predictability** goal.

This example demonstrates that there are limits to interoperability. Sometimes, there are no perfect options. But we view that fact as a challenge, not a dealbreaker. While schema evolution tools can’t magically resolve certain fundamental differences, they can still maximize compatibility across applications, stretching the ability to collaborate as far as possible. Of course, if a group of collaborators wants to achieve closer compatibility, they still have the usual option of all switching to the same software version. Schema evolution tools simply expand the possibility space.

We think it would be worthwhile for future work to explore this tradeoff space in more depth, and explore how it manifests in the context of different applications.

4.0.2 Combining types and migrations makes development easier. Subjectively, we were surprised by how much we enjoyed having both TypeScript types and data translations produced from the same code. The confidence that we had updated every reference to old property names or types made development run more smoothly. Similarly, the ability to trust that the data we got off the disk or wire would be compatible with our running code—even as we changed the schema—made introducing new features easier.

4.0.3 Lenses require well-defined transformations. Many apparently simple data transformations have surprising complexity. Imagine a lens that combines a `firstName` and `lastName` fields into a single `fullName`. This lens works reliably in one direction. All names already stored in the system could be combined into a single field, but there are many names which could not be reliably converted back to “first” and “last” names. The result is a so-called lens that can only run reliably in one direction. It might be practical to support transformations which do not succeed on all possible data and produces errors on some input. We have not yet explored these kinds of transformations, but we believe they would be useful.

5 Related Work

Our work builds on prior work in two main areas. The first area is schema evolution: addressing the problem of evolving

APIs and data formats, using a variety of techniques. The second area is bidirectional transformations and lenses, which form the foundation of our solution approach.

5.1 Schema Evolution

Our work was influenced by approaches taken in industrial systems to the problem of schema evolution.

Stripe, a web payments company, built a middleware system into their API server that intercepts incoming and outgoing communication, and translates it between the current version of the system and the client’s requested version [10]. As a result, developers at Stripe don’t need to concern themselves with the idiosyncrasies of old API requests most of the time, because their middleware ensures requests will be translated into the current version. When they want to change the API’s format, they can add a new translation to the “stack” in their middleware, and incoming requests will be translated to the current version.

This system provides useful encapsulation, but also has some limitations. Developers writing migration rules must implement translations by hand and write tests to ensure they are correct. And because Stripe’s system uses dates to order its migrations, it is limited to a single linear migration path. We adopted the general idea of an encapsulated translation layer, while improving on safety guarantees and allowing for nonlinear migration paths.

The Confluent Schema Registry [2] defines rules that help developers maintain schema compatibility in the Kafka messaging system. For example, if a schema needs to be backward compatible, the Schema Registry only allows adding optional fields, so that new code will never rely on the presence of a newly added field.

While this tool can help prevent a team from deploying incompatible schemas, it does so by severely limiting the changes that can be made to schemas: developers who want to preserve compatibility cannot add new required fields or rename existing fields. Furthermore, these rules limit the guarantees provided by the schema; a record full of optional fields is hard to use, and code that processes it may resort to shotgun parsing, testing for the presence of each field before using it. We aimed to provide a similar level of safety guarantees to developers, while simultaneously allowing for more flexibility.

5.2 Bidirectional Transformations

Bidirectional transformations are a class of “mechanism[s] for maintaining consistency of two (or more) related sources of information.” [3]. These ideas have been applied in a variety of domains, including relational databases, GUIs, and data synchronization.

Lenses are a particular kind of bidirectional transformation, originally developed by Foster et al. to help synchronize

data across different schemas [4].⁴ Later publications by various authors developed many variants of lenses, and we incorporated two particular ideas into our system:

- Symmetric lenses [5] introduced the idea of lenses between structures where each structure may contain information not present in the other. This is in contrast to most previous work, which usually treated one structure as “primary” and another as a “view” with a limited subset of information. The symmetric formulation is a good fit for our use case, since two data schemas might each contain fields not present in the other.
- Edit lenses [6] introduced the idea of transforming edit operations between structures, rather than transforming entire states. This idea helped make the system more efficient; when data changed, we could just transform the latest edit rather than transforming the entire state. The edit-based formulation also combined naturally with the Automerge CRDT, which already stores a log of edit operations.

6 Future Work

As an early prototype, our implementation omits many features that could be added without fundamental changes:

- Adding more lens operators would broaden the scope of possible transformations—including sorting and grouping data, and performing certain kinds of common value conversions like format/unit conversions and math calculations.
- Support for recursive data schemas could be added.
- It might be useful to define a better backward compatibility strategy for the lens language itself, so that old clients can reliably execute lenses created by newer clients.

In addition, the following open questions might result in deeper changes to the model.

6.0.1 CRDT semantics across lenses. Consider the following scenario. There is a scalar value s and an array value a , mapped to one another with the wrap lens, which preserves the consistency relation that s contains the first value in the list a . Both values are CRDTs, as in our Cambria Automerge integration. Now, two users concurrently insert into the first position of a . What happens?

The list CRDT has some deterministic conflict resolution strategy, e.g. arbitrarily putting one user’s insertion first based on their user ID. However, the scalar value also might have its own conflict resolution strategy, e.g. keeping the last write based on an approximate wall clock time [13]. If we run

⁴Some readers may be familiar with lenses in Haskell and other functional languages. Those lenses are based on this same foundational work, but generally focus on getting and setting individual values within a complex data structure, rather than synchronizing two complex data structures with one another.

both conflict resolution strategies independently, there is no guarantee that they will agree in such a way that preserves the desired consistency relation. In order to preserve the consistency relation, it seems that we must align conflict resolution behavior across multiple representations.

We have not yet fully explored the behavior of our system in the presence of concurrent edits and conflicts, but we think that exploring conflict resolution across related data representations is a ripe area for study.

6.0.2 Augmenting data. A change to a schema may require new data, or remove previously required data. Although Cambria has support for providing default data in the case of added or removed fields, it does not provide a mechanism to look up missing data, and so cannot support this kind of change today. To solve this problem, Cambria needs a way to express dependencies on other data sources, or perhaps some kind of callback to allow a lens author to fetch data from other sources.

Similarly, moving data between documents poses challenges. Splitting one document into many, or migrating data from one document to another, are obvious use cases here. Past systems like the XML-transforming language XSLT have struggled with combining and splitting documents.

7 Conclusion

Life on Earth first developed hard structures during the Cambrian period. Our work on local-first software [9] with decentralized data needed more structure, and that was the original inspiration for this project. Before Cambria, we spent substantial time trying to support even small differences in data structures between differing versions of our local-first software.

Today, much of the software we use is monolithic. It hoards data and is unavailable for user modification. We believe software should be more diverse and better adapted to the people who use it. The migration of data from one application to another is often poorly supported or impossible, and real-time collaboration between programs is rare. We accept that to collaborate, we must standardize on a single program like Github Issues or Trello. Perhaps a lens-based system could serve as the glue for connecting these systems. We believe that these tools can be usefully applied not only to schema *evolution*—adapting between versions of the same software—but also to schema *mapping*, adapting related data schemas across different applications.

As with the evolution of organic life, Cambrian software would be free to evolve in different directions while preserving some compatibility. Programs could use lenses to maintain collaboration as they grow. Users and teams could adopt software that suits their needs, rather than standardizing on the lowest common denominator.

Acknowledgments

Thank you to Martin Kleppmann, Adam Wiggins, and Herb Caudill for helpful feedback on this work that greatly improved it. Thanks to Christopher Dare for help with editing the paper.

References

- [1] José Andany, Michel Léonard, and Carole Palisser. 1991. Management Of Schema Evolution In Databases. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB '91)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 161–170.
- [2] Confluent. [n.d.]. Schema Registry Overview — Confluent Documentation. <https://docs.confluent.io/platform/current/schema-registry/index.html>.
- [3] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations*, Richard F. Paige (Ed.). Vol. 5563. Springer Berlin Heidelberg, Berlin, Heidelberg, 260–283. https://doi.org/10.1007/978-3-642-02408-5_19
- [4] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems* 29, 3 (May 2007), 17. <https://doi.org/10.1145/1232420.1232424>
- [5] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2011. Symmetric Lenses. *ACM SIGPLAN Notices* 46, 1 (Jan. 2011), 371–384. <https://doi.org/10.1145/1925844.1926428>
- [6] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2012. Edit Lenses. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '12*. ACM Press, Philadelphia, PA, USA, 495. <https://doi.org/10.1145/2103656.2103715>
- [7] Martin Kleppmann. 2012. Schema Evolution in Avro, Protocol Buffers and Thrift. <https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html>.
- [8] Martin Kleppmann and Alastair Beresford. 2018. Automerger: Real-Time Data Sync between Edge Devices.
- [9] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2019*. ACM Press, Athens, Greece, 154–178. <https://doi.org/10.1145/3359591.3359737>
- [10] Brandur Leach. 2017. APIs as Infrastructure: Future-Proofing Stripe with Versioning. <https://stripe.com/blog/api-versioning>.
- [11] Meredith Patterson, Sergey Bratus, and Dan Hirsch. 2013. Shotgun Parsers. <https://www.youtube.com/watch?v=XVZrmp5MAas>.
- [12] D Jason Penney and Jacob Stein. 1987. Class Modification in the Gemstone Object-Oriented DBMS. (1987), 7.
- [13] Nuno Preguiça. 2018. Conflict-Free Replicated Data Types: An Overview. *arXiv:1806.10254 [cs]* (June 2018). [arXiv:1806.10254 \[cs\]](https://arxiv.org/abs/1806.10254)
- [14] John F. Roddick. 2009. Schema Evolution. In *Encyclopedia of Database Systems*, LING LIU and M. TAMER ÖZSU (Eds.). Springer US, Boston, MA, 2479–2481. https://doi.org/10.1007/978-0-387-39940-9_1532
- [15] Andrea H. Skarra and Stanley B. Zdonik. 1986. The Management of Changing Types in an Object-Oriented Database. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86)*. Association for Computing Machinery, New York, NY, USA, 483–495. <https://doi.org/10.1145/28697.28747>
- [16] Peter van Hardenberg and Martin Kleppmann. 2020. PushPin: Towards Production-Quality Peer-to-Peer Collaboration. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, Heraklion Greece, 1–10. <https://doi.org/10.1145/3380787.3393683>
- [17] Vincent Ventrone. 1991. Semantic Heterogeneity as a Result of Domain Evolution. *ACM SIGMOD Record* 20, 4 (Dec. 1991), 16–20. <https://doi.org/10.1145/141356.141359>