Baseline: Operation-Based Evolution and Versioning of Data

Jonathan Edwards[a] ⓘD and Tomas Petricek[b] ⓘD

a   Independent, Boston, MA, USA

b   Charles University, Prague, Czechia

Abstract   Baseline is a platform for richly structured data supporting change in multiple dimensions: mutation over time, collaboration across space, and evolution through design changes. It is built upon *Operational Differencing*, a new technique for managing data in terms of high-level operations that include refactorings and schema changes. We use operational differencing to construct an operation-based form of version control over data structures used in programming languages and relational databases.

This approach to data version control does fine-grained diffing and merging despite intervening structural transformations like schema changes. It offers users a simplified conceptual model of version control for ad hoc usage: There is no repo; Branching is just copying. The informaton maintained in a repo can be synthesized more precisely from the append-only histories of branches. Branches can be flexibly shared as is commonly done with document files, except with the added benefit of diffing and merging.

We conjecture that queries can be *operationalized* into a sequence of schema and data operations. We develop that idea on a query language fragment containing selects and joins. Operationalized queries are represented as a *future timeline* that is speculatively executed as a branch off of the present state, returning a value from its *hypothetical future*. Operationalized queries get rewritten to accommodate schema change "for free" by the machinery of operational differencing.

Altogether we develop solutions to four of the eight challenge problems of schema evolution identified in a recent paper.

Context: Many technologies have been developed to manage data, yet we still lack convenient general purpose tools for schema evolution and version control of data.

Inquiry: Can the problems of schema evolution and data version control be solved with an operation-based technique? We build upon research on Schema Modification Operations and Operational Transformation.

Approach: Most tools for data management are state-based, for pragmatic reasons of interoperability. We are exploring what is possible with an operation-based approach.

Knowledge: Operational Differencing is able to do fine-grained version control of data even through intervening schema changes. It can be used to synthesize more precisely the information traditionally maintained in a repo. Version control without a repo is simpler and more flexible for ad hoc usage.

Grounding: In prior work with others we identified eight challenge problems of schema evolution in interactive programming systems; We solve four of them.

Importance: Our technique improves upon the known methods for schema evolution and version control of data. Our conjecture that queries can be operationalized could open up a new space of language designs.

The Art, Science, and Engineering of Programming
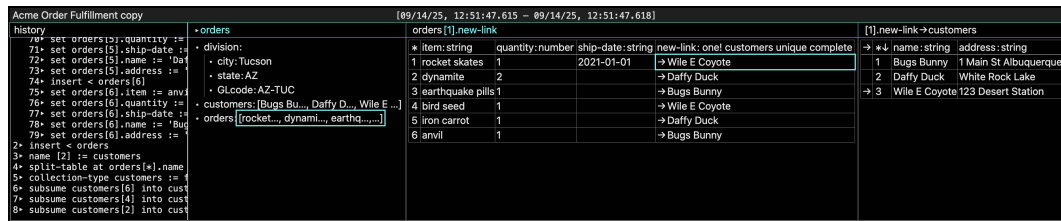
Perspective   The Art of Programming

Area of Submission Database programming, Visual and live programming, Programming environments, Schema evolution, Version control

Baseline: Operation-Based Evolution and Versioning of Data

■ **Figure 1** The Baseline UI

1 Introduction

Data changes in multiple dimensions. Data changes over time, indeed often that is the whole point. Data changes over space between replicas and variant copies. Data also changes in meaning: as needs and understanding evolve so must the shape and interpretation of the data. In practice we handle these different dimensions of change with different technologies embodying different abstractions. Programming languages, file systems, and databases all have distinct ways of managing data change over time. There are many different techniques for distributing, replicating, and collaborating on data, although surprisingly few for version control of data. There is a plethora of tools to assist in evolving the schema of databases, but in practice it often requires ad hoc programming.

Baseline is a research project to develop a general purpose mechanism for data change in all these dimensions, and to provide it conveniently to both programs and humans. To ground this research we first surveyed one of the most problematic aspects of data change: schema evolution. In *Schema Evolution in Interactive Programming Systems* [28] we and others formulated a suite of challenge problems across a range of contexts, including traditional database programming as well as live front-end programming, model-driven development, and collaboration in computational documents. These problems have guided our work and are used as examples in the rest of the paper.

Systems that manage change split into *state-based* and *operation-based* approaches [37, 45, 59]. State-based approaches observe only the before and after of changes, while operation-based approaches record the execution of a set of possible operations. The benefit of state-based approaches is that since they observe from the outside they more easily interoperate with existing tools. The benefit of operation-based approaches is that they can capture user intentions more accurately than can be inferred from the states. For example comparing states of a table schema, a move followed by a rename is indistinguishable from a delete followed by an insert, but that makes a big difference to the data.

In this paper we introduce a new technique for operation-based change management: *Operational Differencing,* which we define on a *rich data* model curated from the features of programming languages and databases.

Baseline manifests all operations in a direction manipulation UI (Figure 1) that gives users the same power as programs. It is difficult to convey the experience of an interactive interface in a written narrative, so we offer the reader a playable demo at

https://thebaseline.dev/Prog26submission/. The UI and playable demo are discussed in Appendices C and D.

The contributions of this paper are:

1. We introduce *Operational Differencing* by example in the familiar setting of typed nested lists and records supplied with a rich set of operations including refactorings. We define the primitive transformations *projection* and *retraction* and the rules governing them.

2. Using these primitives we reconstruct the key features of version control: *diffing* and *transfer* (a specialization of merging more like cherry-picking). Because our technique is operation-based we are able to do fine-grained diffing and merging despite intervening structural transformations.

3. We develop a simplified form of version control without a repo: artifacts under version control have an append-only history of operations attached to them. Branching is just making a copy of the artifact with its history, which can then diverge. There is no need for a repo to record a graph of branches and merges because that can be synthesized from the histories on need, and can be done so optimally in a well-defined sense. By liberating branches and versions of artifacts from the repo they can be shared freely and independently, much like the ubiquitous folk practice of file copying [8, 11], but with the added benefit of being able to diff and merge.

4. We enrich the data model to incorporate relational databases, including operations for schema change and normalization. We demonstrate version control on databases that works across schema changes, and can also be used to manage schema migrations.

5. We conjecture that queries can be *operationalized* into a sequence of schema and data operations, in a form of Programming by Demonstration [23]. We develop this idea on a query language fragment containing selects and joins. Operationalized queries are represented as a *future timeline* that is speculatively executed as a branch off of the present state, returning a value from its *hypothetical future*. This conjecture could open up an unexplored space of query language designs.

6. Operationalized queries have a perhaps surprising benefit: query rewriting [22, 35] during schema change falls out "for free" from the machinery of operational differencing.

7. Altogether we develop solutions to four out of eight challenge problems of schema evolution [28].

2 Rich Data Operations

We start with familiar data structures: nested lists and records. The atomic values are strings, supplied in the set S, and numbers, supplied in N, including `NaN`. What is unusual about this data model is that we assign permanent unique identifiers (IDs) to every record field and list element. These IDs are supplied from the disjoint sets F for record fields and E for list elements. Because record fields have unique IDs their names are only for human readability and we elide them in most examples.

Another unusual feature is that deletion of a record field or list element leaves behind a *tombstone* (see Appendix A).

The data model is typed similarly to statically typed Functional Programming (FP) language such as ML, as follows. List elements all have the same type as defined in the type of the list (tombstones are a bottom type). Record values have the same sequence of fields with the same ID and type of value as in the type of the record. The elements of a list and the fields of a record must have different IDs. The empty record {} serves as a unit type. Every type has an *initial value*. We omit sum types and type aliases because they are not needed in this paper. Figure 2 defines the syntax and operations of the model.

type	value	initial value	
$T ::=$	$v ::=$	$T^{\varnothing} =$	
String	S	""	string
Number	N	NaN	number
List T	$[E{:}\, v\ \ldots]$	$[]$	list
$\{F\ S{:}\ T\ \ldots\}$	$\{F{:}\, v\ \ldots\}$	$\{F{:}\, T^{\varnothing}\ \ldots\}$	record
$\bot$	$\times$	$\times$	tombstone

Value operations	
p noop	Do nothing
p write v	Write atomic value v at p
p insert E before E'	Insert element ID E into list at p in front of element E'
p append E	Append element ID E to end of list at p
p delete E	Delete element ID E in list at p
p move p'	List element p is copied to p' in same list then p is deleted

Type operations	
p Define T	Define p to have type T, initializing values
p Convert T	Convert atomically-typed p to atomic type T
p Rename S	Rename record field p to string S
p Insert F before F'	Insert field ID F into record at p in front of field F'
p Append F	Append field ID F to end of record at p
p Delete F	Delete field ID F in record at p
p Move p'	Record field at p is copied to p' in the same record or containing/contained record, and then p is deleted
p ListOf	Convert value at p into a list of one element with ID $\mathbb{1}$
p IntoFirst	Convert list at p into its first element, else the initial value
p RecordOf F	Convert value at p into a record of one field with ID F
p IntoField F	Convert record at p into value of its field F

■ **Figure 2** Rich data syntax and operations

A *path* p is a possibly empty sequence of IDs denoting a path drilling into nested records and lists. Paths can access both values and types. We separate the IDs in a path with dots, and a single dot is the empty (top) path. A special element ID $*$ is used to access the element type of a list type.

A *state* pairs a value with its type. We abuse the notation $v :: T$ to both denote a state and assert that the value matches the type. An *operation* is a partial function from an *input state* to an *output state*. Operations are not defined on all input states, imposing various preconditions. All operations take a *target* path as a parameter indicating that the operation is to be performed at that path within the state. Operation use infix syntax, placing the target path first, then the operation name, followed by any other parameters.

We execute operations with the $\mathring{,}$ infix operator taking an input state on the left, an operation on the right, and yielding an output state which may be chained into subsequent executions. Here is an example that starts with an empty list of numbers and adds an element with ID e and value 1:

$$S_1 = [\,] :: \mathsf{List\ Number}$$
$$S_2 = S_1 \mathbin{\mathring{,}} \;.\;\mathsf{append}\;e\mathbin{\mathring{,}}\;.e\;\mathsf{write}\;1$$
$$\quad = [e\colon 1] :: \mathsf{List\ Number}$$

The append targets the empty path $.$ referring to the whole list. Then the write targets the new element with the path $.e$. In all our examples we take IDs like e as given but in practice the Baseline API generates them uniquely.

A *table* is a list of records, the fields of which are the *columns* while the elements of the list are the *rows*. Here is an example of changing the type of a table:

$$S_1 = [e\colon \{x\colon 1,\ y\colon 2\}] :: \mathsf{List}\{x\colon \mathsf{Number},\ y\colon \mathsf{Number}\}$$
$$S_2 = S_1 \mathbin{\mathring{,}}\;.*\;\mathsf{Append}\;z \mathbin{\mathring{,}}\;.*.z\;\mathsf{Define\ Number}$$
$$\quad = [e\colon \{x\colon 1,\ y\colon 2,\ z\colon \mathsf{NaN}\}] :: \mathsf{List}\{x\colon \mathsf{Number},\ y\colon \mathsf{Number},\ z\colon \mathsf{Number}\}$$

The Append operation targets the path $.*$ which is the record type of the list's elements and adds a new field z to it, in effect adding a column to the table. The Define operation targets the path $.*.z$ which is the type of the new field (initially the unit type {}) and changes it to Number. The new field is also inserted into all elements of the list with the initial value NaN.

The append and write operations are *value operations*, meaning they do not modify the type of the state. On the other hand Append and Define are *type operatons* that may modify both the type and value. Type operations do what databases call *schema migration*: the value is adapted to match the type while minimizing loss of information, iterating over list elements as needed (think of the $*$ in the type path as a wildcard). By convention we capitalize the name of type operations. Baseline has a *user mode* in which only value operations are permitted.

Remark. In this presentation we adopt the standard approach of defining values and types as distinct objects. However Baseline only has values, with initial values serving as prototypes. Every list contains an initial *header* element with the ID $*$ containing an initial value which serves as the prototype of the list elements. Note how this convention corresponds to the visual rendering of a table where a header row describes the type of each column. Types are a powerful abstraction for the theory and implementation of programming systems but we conjecture that they can be replaced with prototypes, without loss of power, to simplify the programming experience. We have also explored an untyped version of our approach [55].

Many of the type operations exist in order to do schema migration on values: they would not be needed just to define a type prior to populating it with data. We think of these operations as *type refactorings*, capturing high-level design changes. They are manifested in the UI as direct manipulations so that schema evolution can be done interactively. Schema evolution is also needed in *live programming* [28].

$S_1 = [e: \{\mathit{what}: \text{"clean"}, \mathit{who}: \text{"Jack"}\}] :: \text{List}\{\mathit{what}: \text{String}, \mathit{who}: \text{String}\}$

$S_2 = S_1 \; \mathbin{\text{\fontfamily{cmr}\selectfont⨾}} \; .*.\mathit{who} \; \text{ListOf}$

$\quad = [e: \{\mathit{what}: \text{"clean"}, \mathit{who}: [\mathbb{1}: \text{"Jack"}]\}] :: \text{List}\{\mathit{what}: \text{String}, \mathit{who}: \text{List String}\}$

■ **Figure 3** TODO refactoring

For example a common design evolution is when a single value needs to become a list of multiple values. We capture this in the ListOf operation, shown on a TODO table in Figure 3. The ListOf operation takes any type and wraps it inside a list, and wraps all corresponding values into single-element lists using the element ID $\mathbb{1}$. We will return to this example in the next section.

3 Operational Differencing

Change management systems deal with what happens when two copies/replicas-/branches of something diverge. Among these systems there is a dichotomy between *state-based* and *operation-based* techniques [37, 45, 59]. In this section we introduce a new form of operation-based change management: *Operational Differencing*.
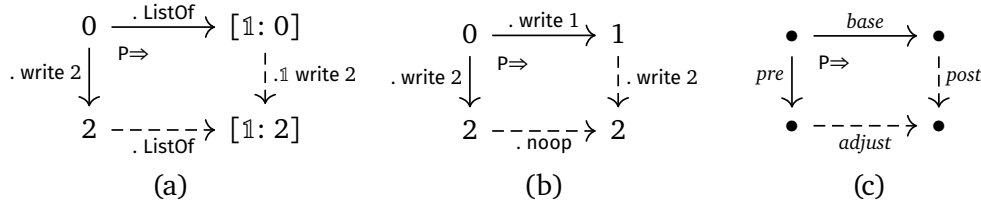
The benefit of operation-based techniques is that they can capture more information than can be inferred from the states. For example a Rename operation makes many changes throughout the state, and standard version control systems will see these changes as unrelated and conflicting with any other changes on the same text lines. In contrast operational differencing sees a renaming as a single change that conflicts only with renaming to a different name. The benefit of state-based techniques is that they can observe other systems from the outside, which may explain why all existing version control systems are state-based.

Remark. We discuss source code version control systems like git as a point of reference, but we do not propose to compete with them. Git is highly adapted to the needs

of intensive development in the Unix environment and the cultural norms of Unix developers. In that context version control must be state-based in order to interoperate with existing tools and practices. Instead of source code we focus on rich data, where we believe operation-based version control offers important benefits.

We define a *timeline* to be an initial state and a sequence of operations executed in order that are each valid on their input state, producing an end state. A *history* is a timeline starting from the initial empty state of a document, which by convention is the unit value. Baseline records histories by observing the operations executed in the API and in a UI that manifests operations as direct manipulations. Operational differencing is built upon two primitive functions on operations: Project and Retract.

3.1 Projection

$$
\begin{array}{ccc}
0 \xrightarrow{.\ \mathsf{ListOf}} [\mathbb{1}\colon 0] & \quad 0 \xrightarrow{.\ \mathsf{write\ 1}} 1 & \quad \bullet \xrightarrow{base} \bullet \\
\;\;\;\;\;\;\; \mathsf{P}\!\!\Rightarrow & \;\;\;\;\;\;\; \mathsf{P}\!\!\Rightarrow & \;\;\;\;\;\;\; \mathsf{P}\!\!\Rightarrow \\
{\scriptstyle .\ \mathsf{write\ 2}}\Big\downarrow \quad\quad \Big\downarrow{\scriptstyle .\mathbb{1}\ \mathsf{write\ 2}} & {\scriptstyle .\ \mathsf{write\ 2}}\Big\downarrow \quad\quad \Big\downarrow{\scriptstyle .\ \mathsf{write\ 2}} & {\scriptstyle pre}\Big\downarrow \quad\quad \Big\downarrow{\scriptstyle post} \\
2 \dashrightarrow[{\scriptstyle .\ \mathsf{ListOf}}]{} [\mathbb{1}\colon 2] & 2 \dashrightarrow[{\scriptstyle .\ \mathsf{noop}}]{} 2 & \bullet \dashrightarrow[{adjust}]{} \bullet \\
(a) & (b) & (c)
\end{array}
$$

■ **Figure 4** Projection

The Project function takes as inputs two timelines with the same initial state and produces two new timelines continuing from the input timelines and converging on the same final state. It is easier to understand this with diagrams. In Figure 4(a) the arrow on the left takes the state 0 (we elide the obvious type signature) and writes a 2 over it. The arrow on the top wraps that 0 into a list with one element $[\mathbb{1}\colon 0]$. The projection function takes these two arrows as input and produces the two dashed arrows such that the diagram *commutes*, meaning the dashed arrow operations are valid on the states at their bases, and they both produce the same state $[\mathbb{1}\colon 2]$. The way this is achieved is to map the target path of the write operation through the ListOf operation. We say that the two write operations have the same *intention*—they "do the same thing"—they convert the 0 to a 2, even though the 0 has moved to a different location. That is the key principle of projection: it produce a right-hand operation that "does the same thing" as the left-hand operation despite the top operation having intervened.

Figure 4(b) shows a messier situation. Here the left and top operations are both writing different values to the same location. This is called a *conflict*. The rule is that projection attempts to preserve the intention of the left operation, even if that means overriding the top operation, which as a consequence gets converted to a noop on the bottom.

This last example shows that projection is asymmetric: flipping the diagram on its diagonal to switch left and top yields a different result. And in fact we will have occasion to flip and rotate these diagrams, so we cannot depend on terminology like "left" and "top". Figure 4(c) shows our orientation-neutral terminology: projection

converts the *pre* operation into the *post* operation, preserving its intent after the *base* operation intervenes. The *base* operation is converted into the *adjust* operation preserving as much of its intention as is allowed by the first rule. We also graphically indicate the orientation of the diagram by placing P$\Rightarrow$ into the corner of the initial state.


(a) (b)
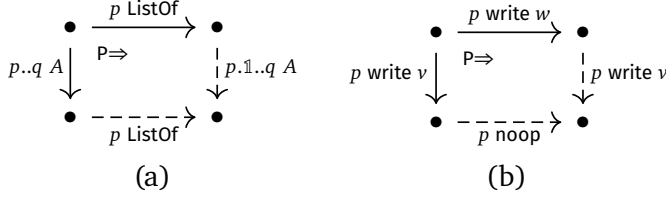
Figure 5 Projection rules

Projection is defined with rules that can be diagrammed similarly. Figure 5(a) is a rule covering the ListOf example. The target path of the ListOf is abstracted to p, and the write operation is abstracted into any operation A targeting p or deeper, which is indicated with the path concatenation operator in $p..q$. Operation A is projected down into the ListOf with the path $p.\mathbb{1}..q$.

Figure 5(b) is the rule for conflicting writes.

3.2 Retraction

(a) (b) (c)
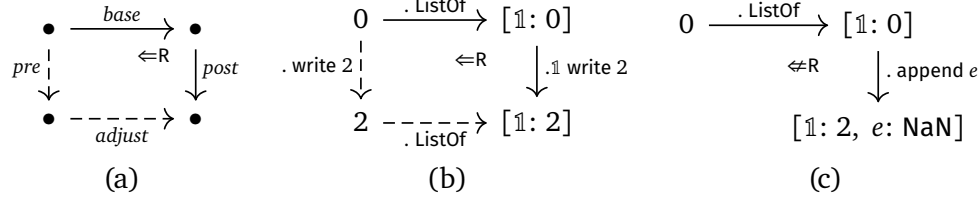
Figure 6 Retraction

The Retract function goes in the opposite direction of Project. Figure 6(a) shows that it starts with two timelines *base* and *post* and produces *pre* and *adjust*. We say that *post* is retracted through *base* into *pre*. The goal as with projection is that *pre* preserves the intention of *post* before *base* happened—it "does the same thing". Likewise *adjust* preserves the intention of *base* but defers to the first rule.

Figure 6(b) shows the retraction reversing the projection in Figure 4(a). Often retraction is the inverse of projection as in this case, but not always, and sometimes retraction is not possible at all. In Figure 6(c) the list has a new element appended to it. There is no way to do the same thing before the list was created. Projection failures indicates a *dependency* between operations. The append operation depends on the prior ListOf operation having created the list and cannot be retracted through it.

Retraction is defined with rules like those for projection so we skip over them in the interest of brevity.

Note. We have implemented projection and retraction rules with if and switch statements, which has become impractical as the number of operations has grown, so much so that our implementation is still incomplete. We plan to build a more succinct DSL that exploits the inherent symmetries in the rules: projection is often symmetric in the two input operations; projection and retraction are often inverses.

Remark. How do we know if the rules for projection and retraction are correct, that they do indeed preserve the intentions of operations? Our answer is that there is no objective answer to this question. The rules are essentially axioms defining what it means to preserve intention. Well-designed rules will agree with intuition and avoid surprises. One of these intuitions is that transfer should be symmetric: transfering an operation across a diff and then transferring it back again should bring back the same operation. We call this the *roundtrip law* and we test that our rules obey it. Except there are exceptions, like when an operation gets dropped because its target was deleted. We keep discovering new exceptions that require weakening the law, so in future work we hope to come to a more principled understanding.

3.3 Across the Multiverse

A *diff* between two states A and B is a pair of timelines that branch off some shared state O to yield A and B, diagrammed like this:

$$A \xleftarrow{a_n} \quad \xleftarrow{\quad} \quad \xleftarrow{a_1} O \xrightarrow{b_1} \quad \xrightarrow{\quad} \quad \xrightarrow{b_m} B$$

The two timelines are called the *branches* of the diff and we call their operations the *differences*. Many version control systems use something like a diff by looking for the *last common ancestor* in a tree of branching versions, which is then used for *three-way diffing* [45]. We do not do that—we will discuss how to calculate a diff in §3.4 but for the moment let us take them as given. The key operation on a diff is to *transfer* an operation from one side to another, which we define diagrammatically in Figure 7 (where we lift projection and retraction from operations to timelines in the obvious way).

$$\text{Transfer}(a_{1\ldots n}, O, b_{1\ldots m}) = b' \text{ in}$$

$$\begin{array}{ccccc}
A_{n-1} & \xleftarrow{a_{n-1}\ldots a_1} & O & \xrightarrow{b_1\ldots b_m} & B \\
a_n \downarrow & \text{R}\Rightarrow & \downarrow o \quad \text{P}\Rightarrow & & \downarrow b' \\
A_n & \xleftarrow{\quad adjust_a \quad} & O' & \xrightarrow{\quad adjust_b \quad} & B'
\end{array}$$

Figure 7 Transfer

Transfer[1] retracts a_n backwards through all earlier operations on the A timeline onto the original state O and then projects that operation forwards through all the operations of the B timeline. The final result is the operation b' that yields a new version B'. The effect of the transfer is that the final operation b' preserves the intention

---

[1] Transfer was called migrate in previous publications [26, 27].

of the original operation a_n given all the operations traversed while traveling backward in time to O and then forward to B. Transfer also computes a new diff between A_n and B': the transferred operation is now shared between them in O' with branching timelines $adjust_a$ and $adjust_b$. This updated diff is used later for diff optimization (§3.4).

In this definition we transferred the last operation of the A branch—an earlier operation can be transferred by using the corresponding truncation of A. Since transfer uses retraction it may fail because of a dependency on an earlier operation in the A branch. However it is always possible to transfer any operation by first transfering all of its dependencies earlier in the branch. A *sync* action that transfers all operations in the branch will always succeed. We also provide a finer-grained *dependency transfer* function that transfers just the transitive dependencies and so always succeeds.

As an example we extend the TODO refactoring in Figure 3 into two branches: branch A reassigns the TODO from Jack to Jill; branch B converts the assignment into a list, changes the assignment from Jack to Jacque and inserts a new assignment to Tom:

$$O = [e: \{what: \text{"clean"}, who: \text{"Jack"}\}]$$
$$a_1 = .e.who \text{ write "Jill"}$$
$$A = [e: \{what: \text{"clean"}, who: \text{"Jill"}\}]$$
$$b_1 = .*.who \text{ ListOf}$$
$$b_2 = .e.who.\mathbb{1} \text{ write "Jacque"}$$
$$b_3 = .e.who \text{ insert } g \text{ before } \mathbb{1} \mathbin{\mathring{,}} .e.who.g \text{ write "Tom"}$$
$$B = [e: \{what: \text{"clean"}, who: [g: \text{"Tom"}, \mathbb{1}: \text{"Jacque"}]\}]$$

We can transfer these operations individually from one side to the other. If we transfer b_2 into A the change from Jack to Jacque gets applied to the who field (showing the change in bold):
$A' = [e: \{what: \text{"clean"}, who: \textbf{"Jacque"}\}]$. If we transfer a_1 into B the change from Jack to Jill gets applied to the list element containing Jack, even though it has been wrapped and shifted:
$B' = [e: \{what: \text{"clean"}, who: [g: \text{"Tom"}, \mathbb{1}: \textbf{"Jill"}]\}]$.

This example demonstrates how transference can propagate changes bidirectionally through structural changes. Standard state-based version control techniques would see only a big conflict and require a human to pick out and propagate finer-grained changes correctly. Operational differencing does this more precisely because it tracks how locations shift through time.

Version control systems like git provide a variety of actions like *merge*, *rebase*, and *cherry-pick* with subtly different semantics. We provide transferrance as the sole primitive and then build coarser-grained actions out of sets of transfers. For example all the changes in one branch can be transferred to the other. Just the type changes or just the data changes can be transferred. The dependency transfer action mentioned earlier transfers all transitive dependencies. To *synchronize* two branches all changes are transferred from one to the other and then all changes remaining in the other

branch are transferred back. Note that synchronization has a direction: syncing from *A* into *B* resolves all conflicts in favor of *A*.

3.4 Diff Optimization and Synthesis

We have been discussing transference assuming we are given a diff that specifies a shared state and timelines branching off of it. The intuition is that the shared state should contain everything the two final states have in common, and the branches should represent the differences. Not all branching timelines will satisfy that intuition. There might be redundant operations duplicated in both branches, either by coincidence or by explicit transfers. There could also be redundant changes within a branch that override earlier ones. All of these sorts of redundancies are undesirable because they add noise to a visualization of differences, and in some cases can cause transfer to produce surprising results. There is a simple algorithm that can be used to produce optimal diffs using transference itself.

Note that in Figure 7 the bottom of the diagram produces a new diff: two new timelines $adjust_a$ and $adjust_b$ branching from a new shared state O'. These *adjusted* timelines eliminate the redundancy of the original operation and its transferred image. Clearly after executing a transfer we should drop the original diff and use the adjusted diff instead. In the same way we can use transfers to remove redundancies in diffs as follows. Given a pair of branching timelines we iterate over all the actions on both sides in order.[2] For each operation we try to transfer it to the other branch. If the transfer succeeds and its result is a fixpoint (it doesn't alter the state of the other side), then we discard the branches in favor of the adjusted ones and continue the loop. The intuition here is that if the transferred operation makes no difference to the other side then it isn't really a difference, so we append it to the history of the shared state. After we have tried every transfer from both sides the result will be optimal in the sense that the history of the shared state is maximal while the timelines branching off of it are minimal, without redundancies. Note however that this maximal shared state may not have ever actually occurred in either history—it is the *hypothetical* ideal last common ancestor if history had happened in a fortuitously non redundant order. This ideal common ancestor avoids the problems that can be caused by cherry-picking in git [32].

Diff optimization lets us synthesize an optimal diff from the histories of any two states. A history always starts from the unit value as its initial state. So all histories are branches from the initial state and we just optimize that primordial diff.

Note. We left out a detail in the diffing algorithm. Sometimes changes cancel each other out, like an insert followed later by a delete, or an operation followed later by its *undo* (Appendix B). The diff timelines are simplified by attempting to fully retract each operation—if the retraction results in a `noop` it was canceled out and the simplified adjusted timeline is used instead.

---

[2] Currently it does not matter how the two branches are interleaved by this algorithm, but future plans may require a causal ordering on transferred operations.

3.5 Low-fuss Version Control

The ability to synthesize diffs for any two histories avoids the need for a *repo* to determine the last common ancestor for operations like merging. As we have seen that can be sub-optimal as a basis for three-way diffing. But more importantly, understanding and maintaining the repo imposes a large mental burden on the users of version control systems. Git [13] is infamous for causing confusion and frustration [16, 50, 52].

We present the user with a simpler conceptual model: every document has an append-only history. To branch an alternate version of a document, simply copy it. The copies can then diverge, and their histories can be compared at any time to compute their diff, which can then be used to transfer differences between them. We can record copy and transfer events as comments in histories in order to reconstruct a version graph like that maintained in a repo.

This approach is more flexible: since versions and branches no longer belong to a repo they can be shared without sharing the whole repo. They are like files (and likely implemented as files): independent storage units that can participate in overlapping and shifting webs of collaboration. They can be emailed. We regain the flexibility of the ubiquitous folk practice of file copying [8, 11], but with the added benefit of being able to diff and merge.

Remark. Large projects will still need to maintain a list of all branches, but that could be as lightweight as a file directory. Large projects will also want to compress storage and cache diffs. We believe all these capabilities could be added on top of our approach with far less user-facing complexity than git. But that is not an immediate goal: we want to first optimize for small-scale ad hoc usage.

4 Database Evolution

In this section we apply Operational Differencing to the problem of *schema evolution* in relational databases. We do this by adding relational database features into our data types and operations. For example Figure 8a shows a table of orders for the Acme Corporation. A database professional will immediately see a problem: the customer name and address is duplicated across all orders by the same customer, so any changes to that information would need to be duplicated. Database textbooks teach to avoid this sort of problem by properly *normalizing* the schema before adding any data. However in practice the need for schema changes often arises after there is data, which must then be *migrated* into the changed schema, typically with ad hoc SQL programs.

One approach a SQL migration program could take is to populate a new customers table from a query projecting out the name and address columns and excluding duplicates. Then it would need to generate primary keys for the deduped customers (the name cannot be the primary key because it is mutable). A new column customer would be added to orders declared as a foreign key containing the customer's primary key. Finally the name and address columns in the orders table would be dropped.

orders:

ID	item	quantity	ship_date	name	address
e_1	Anvil	1	2/3/23	Wile E Coyote	123 Desert Station
e_2	Dynamite	2		Daffy Duck	White Rock Lake
e_3	Bird Seed	1		Wile E Coyote	123 Desert Station

(a) Single orders table

orders:

ID	item	quantity	ship_date	customer
e_1	Anvil	1	2/3/23	$\rightarrow e_1$
e_2	Dynamite	2		$\rightarrow e_2$
e_3	Bird Seed	1		$\rightarrow e_3$

customers:

ID	name	address
e_1	Wile E Coyote	123 Desert Station
e_2	Daffy Duck	White Rock Lake
e_3	Wile E Coyote	123 Desert Station

(b) Split orders and customers tables

orders:

ID	item	quantity	ship_date	customer
e_1	Anvil	1	2/3/23	$\rightarrow e_1$
e_2	Dynamite	2		$\rightarrow e_2$
e_3	Bird Seed	1		$\rightarrow e_1$

customers:

ID	name	address
e_1	Wile E Coyote	123 Desert Station
e_2	Daffy Duck	White Rock Lake
e_3		

(c) Deduped customers

■ **Figure 8** Table schema change

type	value	initial value	
$\rightarrow p$	$\rightarrow E$	$\rightarrow *$	link to element E of list p ($*$ is null link)

Value operations	
p link E	Link to element E (unlink if $*$)

Type operations	
p Link p'	Define p as link to list p'
p Split p'	Split at column p creating linked table p'
p Join	Join link column p deleteing linked table

■ **Figure 9** Additional database syntax and operations

Baseline: Operation-Based Evolution and Versioning of Data

To perform the migration in Baseline we first need to add support for relational data into our types and operations. We already support tables: they are lists of records, where the elements of the list are the rows and the fields of the record are the columns, which are homogeneously typed as in a relational database. Throughout the rest of the paper we will refer to any Baseline document containing tables as a *database*. What is missing so far is *relationships* between tables, which relational databases model through primary and foreign keys. We take a more PL-oriented approach. Instead of primary keys exposed as user data fields we annotate list elements with IDs that are unique, permanent, and opaque. Instead of foreign keys we add the *link* data type in Figure 9: the type of a link defines the table (or list) it ranges over; the value of a link is an element ID in that range, or $*$ for a null link. We also add the operations defined in Figure 9. All operations that move a list must also adjust the path in the type of all links into that list.

The first step in evolving the Acme database is to Split out a Customers table with the following operations:

$$\begin{array}{ll} \{\textsf{orders: [\ldots]}\} \; \mathbin{\raisebox{0.2ex}{\textsf{\r{,}}}} & (1) \\ \textsf{. Append customers} \; \mathbin{\raisebox{0.2ex}{\textsf{\r{,}}}} & (2) \\ \textsf{. orders.} * \textsf{. name Split customers} \; \mathbin{\raisebox{0.2ex}{\textsf{\r{,}}}} & (3) \\ \textsf{. orders.} * \textsf{. name Rename "customer"} & (4) \end{array}$$

(1) Starting from the state in Figure 8a; (2) The Append operation adds a customers field to hold the new table. (3) The Split operation then splits the orders tables in two starting at the name column, copying those columns into the customers table, the rows of which are given the same IDs as in the orders table. The original name column in orders is replaced with links to the corresponding customers rows. (4) The Rename operation renames the links column to be customer.[3] The result is shown in Figure 8b.

The last step of this database evolution is to de-duplicate the customers, for which we use the move operation from Figure 2. Thus:

$$\textsf{. customers.} e_3 \textsf{ move . customers.} e_1$$

The move operation merges e_3 into e_1, resulting in the state shown in Figure 8c. Note that the order for Bird Seed has had its link to e_3 forwarded to e_1. This completes the schema evolution.

Note. The preceding discussion does not reflect the full complexity of our data model. Lists and tables can be nested. Links are set-valued, and can drill into nested tables. We believe this complexity is justified to cleanly unify PL and DB data models. The Split and Join operations are composed from a set of primitive operations on nested tables and links. We have experimented with several alternative sets of primitives, with subtle tradeoffs on transfer semantics (e.g. how does appending a column get projected through the split operation?). We need further work to settle this design.

---

[3] Abusing notation by conflating the name and ID

4.1 Database Version Control

Schema evolution is typically implemented as a one-time one-way event, which becomes problematic when there are multiple replicas of the database in production. Operational Differencing is more flexible because it can transfer changes forward through the schema evolution and backwards as long as they aren't dependent on the schema evolution itself. This could allow distributed migrations to happen asynchronously. The scheme evolution operations themselves can be transferred between copies of a database, so migrating a database instance can be done by transferring in the type changes from the dev branch instead of an elaborate DevOps procedure.

For example say that the Order Fulfillment department at Acme Corporation is happy with the old version of the database because they don't have to do a query to find the customer name and address. So they don't want to deploy the schema changes made by the Customer Relationship department. They could instead continue to use the old database and transfer updates to the new database. Likewise when a new order is entered in the new database it can be transferred to Fulfillment in its unnormalized form. But new orders to new customers can't be transferred backwards because customer creation is dependent on the Split operation creating the customers table. In those cases the new order will transfer back with blank name and address fields. This problem is an instance of the classic *View Update Problem*[4, 31]. A better solution for this example uses queries as discussed in §5.1.

Even with the limitations on view updates, Operational Differencing brings many of the advantages of version control to the domain of databases. It is possible to calculate the precise differences between two database instances despite intervening schema changes. It is always possible to transfer all the differences from one to the other, and often possible to transfer individual differences.

Note. Above we used a move operation to deduplicate the two coyotes e_1 and e_3. We took that approach because it was the simplest thing that could possibly work, but arguably it doesn't. Say that e_1 changed its address in the old database and we transfer that change forward into the deduped table. Our current implementation drops the change, because e_3 overwrites e_1. We believe a more intuitive result would be to "de-deduplicate" leaving two coyotes with the same name but different addresses. In §7 we propose a set-like datatype to solve this problem. The moral of this story is that it is surprisingly hard to get the transfer semantics of operations right.

5 Operationalized Queries

We saw in the last section that although the Fulfillment department preferred the unnormalized view of orders it would be forced to migrate to the normalized database to stay fully in sync with all changes. Relational databases offer a better solution: *queries*. In this section we discuss how Operational Differencing can build queries and updatable views. But the simplest solution doesn't even require a query. We can just make a copy of the database and denormalize it.

The `Join` operation in Figure is the inverse of `Split`. It replaces a link column with all the columns of the linked table, copying their contents to match the link values.[4] Specifically:

```
.orders.*.customer Join
```

resulting in exactly the database we started with in Figure 8a. The Fulfillment department can perform this operations directly in the UI, allowing them to operate in their preferred schema and bidirectionally sync changes with the normalized primary database in the Cutomer Relationship department. Bidirectional sync works now when it didn't before because the `Join` operation does not create any dependencies. We could make this even more convenient by automatically syncing on all changes. We have essentially built an updatable view by doing schema changes on a copy, and we did this in the UI without programming.

This example raises an interesting question: could we generalize it into a general technique for building queries? Instead of writing a query in an abstract language we make a copy of the database and perform schema change and data mutation operations in the UI to *sculpt* [38] it into the desired form. This sculpted database acts like a query if we automatically sync changes from the primary database into it. Such forward projection is always possible. We can also update the sculpted view and sync those changes back to the real database, limited to those cases where a traditional query-based view would also be updatable.

We call this idea *operationalized queries*. It can be seen as a form of *Programming by Demonstration* [23]. We can draw another analogy: query languages are a classic example of pure functional programming that avoids mutation. Pure functions can often be implemented by copying values, mutating them, then returning the result. That is exactly what we are doing: computing a query by mutating a copy of the database.

It takes more than a join operation to make a query language. We can take another step with a simple selection predicate. When the Fulfillment department ships an order it sets the date in the `ship_date` field. They would like a view showing just the pending orders, which lack a ship date. We can further sculpt the view with the `deletePresent` operation in Figure 10:

```
.orders.*.ship_date deletePresent
```

The target path is the type of the `ship_date` column—all orders with a non-blank value in that column are deleted, producing the desired view. When a ship date is entered into this view the change will transfer back into the primary database, which has the effect of dropping that order from the recalculated view. General purpose queries require a general language of predicates. We could embed a predicate as a Boolean-valued computed column and then use the `removePresent`/`removeAbsent` operations. That would be consistent with the philosophy of "querying by sculpting" but we plan to also explore adding predicates into the language of operations.

---

[4] With the set-valued links discussed earlier it becomes a relational join operation.

type	value	initial value	
$\langle op \,\ifmmode\circ\else\fi\ldots\ \uparrow p\rangle$			Formula returning value at p.
Value operations			
p deletePresent E	Delete rows where column p has non-initial value.		
p deleteAbsent E	Delete rows where column p has initial value.		
Type operations			

■ **Figure 10** Additional query syntax and operations

5.1 Formulas as Future Timelines

We have seen how an auto-synced branch of a database can serve as a query on it, and how we can build that query by sculpting the database in the UI. But that leaves us having to hop around different branches to view queries. It is more convenient to embed multiple queries into the database itself as *formulas*. A field defined as a formula has its value computed by evaluating the formula, like a formula in a spreadsheet cell. Matching the form of operational queries, we define a formula as a type in Figure 10 consisting of a timeline of operations ending with a *return path*. A formula is evaluated by executing the timeline of operations starting with the current state of the database and then returning the final value of the return path. In this sense a formula is a *future timeline*—it is a speculatively executed branch off of the present that returns a value from that hypothetical future.

Formulas as future timelines makes it easy to use Programming by Demonstration [23]. In the previous example we created a query by making a copy of the database and "sculpting" it into the result we wanted with Join and deletePresent operations. The UI lets us view the differences between the original and sculpted databases (Appendix C) and provides a command *Transfer into Formula* that takes the timeline of differences on one branch of the diff and injects them as a formula into the other side. The return path of the formula is set from the location currently selected in the UI. Applied to the previous example after selecting the denormalized orders table it will append a formula to the normalized database with the operations:

```
. Append new-formula ⨾
new-formula Define
    ⟨ .orders.∗.customer Join ⨾ .orders.∗.ship_date deletePresent ⨾ ↑ .orders ⟩
```

Henceforth the *new-formula* field's value will be the result of a query denormalizing and filtering the current contents of the orders table.

5.2 Query Rewriting

There is another benefit to treating queries as timelines of operations: operation transfer gives us *query rewriting* [22, 35] for free. When a database schema changes, some of the queries written for it may need to change too. This scenario is called forward rewriting—backward rewriting is backporting a query on a new schema to an older one. In practice query rewriting is done manually by SQL programmers. The just-cited research adapts query optimization techniques to build specialized rewriting algorithms. Baseline handles query rewriting with the existing machinery of transferrance: the Define operation defining the query is transferred through the schema change operations.

For example, we can transfer the prior example query on the normalized database back into the original unnormalized database we started with in Figure 8a. Doing so retracts the Define operation backwards through the Split operation that normalized the database (ignoring the deduping move because it is a value operation that doesn't affect the schema). Retracting a formula definition simply retracts its timeline of operations. Thus the formula's Join operation retracts through the database branch's Split operation, which cancels out into a noop, which is dropped from the formula. The formula's deletePresent operations retracts without change. The result of the transfer is executed on the original database:

. Append *new-formula* ⸵
.*new-formula* Define
⟨ .orders.∗.ship_date deletePresent ⸵ ↑ .orders ⟩

which defines a query to filter the pending orders from the unnormalized orders table.

Forward rewriting is more complicated. It occurs when a formula definition is transferred into a branch containing type differences (i.e., schema changes). Forward rewriting is also done on all formulas within a database whenever a type operation is executed. To transfer a formula forward through a type operation its *inverse* (Appendix B) is prepended to the formula. The intuition here is that the query was written to apply to the old schema, so to preserve its meaning in the new schema we must first convert the schema back again.[5] For example if we were to transfer the prior backported query forward again through the Split operation then the inverse operation Join would be prepended to the deletePresent operation, correctly reconstructing the original query.

6 Evaluation

We evaluate Baseline against the challenge problems proposed in *Schema Evolution in Interactive Programming Systems* [28].

---

[5] This naive algorithm would clutter up the formula with irrelevant schema changes. We simplify them out by retracting the original formula back through them as much as possible, retaining only those schema changes the formula is dependent upon.

1. **Live State Type Evolution** ✘ The problem is to evolve the state of an Elm application so that it needn't be restarted. The example given is to evolve a field of a TODO item from completed: bool to completed: Maybe<DateTime>. This example raises the question of how Baseline handles optional types, but it is moot. Baseline does not yet have enough programming capabilities to implement the TODO application.

2. **Extract Entity** ✔ We adopted this problem as the example shown in Figure 8 and solved it in §4 with the Split and move operations.

3. **Code Co-evolution for Extract Entity** ✔ This problem builds upon the previous one, asking that a query for pending orders be rewritten during the schema evolution. We adopted this example and solved it in §5.2, although only in a fragmentary query language.

4. **Structured Document Edits** ✘ The problem is to take string data in a table column and parse it into multiple columns. Changes should be transferable bidirectionally between the new document and a variant of the old. We have implemented a solution using a parse operation taking a regular expression that splits a string column of a table. But we judge that regular expressions violate Baseline's principle that operations are concrete direct manipulations, so we disclaim a solution.

5. **Code Co-Evolution for Structured Document Edits** ✘ This problem builds upon the previous one, asking that formulas be rewritten during the schema evolution. Baseline does not yet have enough of a formula language to meet the requirements of this problem.

6. **Divergence Control for Extract Entity** ✔ This problem extends problem #2, requiring that changes be bidirectionally transferable through the schema evolution. We solved it in §4.1, albeit with the well-known limitations of view updates.

7. **Multiplicity Change in Schema** ✔ This problem is about allowing different versions of code to interoperate bidirectionally even though they have different schema, specifically a schema evolution converting a scalar value into a list. We adopted this problem as the example shown in Figure 3 and solved it in §3.3. An extended version of this problem is posed by Litt et al. [48] (Appendix III) involving null values that we can't handle yet.

8. **Live Editing Domain-Specific Languages** ✘ This problem is the most challenging. It asks to evolve data and correspondingly adapt the internal execution state of code concurrently operating on it. This appears intractable in general, but the problem is framed in a state-machine DSL where it may be possible. Baseline lacks enough programming capability to handle this problem yet.

Altogether we solved four of the eight challenge problems. We keep the remaining four as goals for future work. Appendix E contains an evaluation of Baseline using the *Technical Dimensions of Programming Systems* [42] framework.

7   Discussion

There is an abundance of problems for future work.

Baseline assigns a unique ID to every insertion operation. Sometimes that might be undesirable, as when people working in different branches coincidentally make the same changes. Diffing will treat their insertions as different. This is one case where state-based version control has a more intuitive result. The diffing UI should detect such cases of adjacent similar insertions and offer to coalesce them (with move operations).

Implementing projection and retraction rules with if statements has proven impractical. We want to build a DSL for specifying rules more succinctly, exploiting their inherent symmetries. We want to use property-based testing to check that the rules obey the *roundtrip law*, but first we need to better understand exactly what that law minimally requires.

Our simple approach to deduplication is flawed. We plan to implement a proper set datatype that retains provenance by collapsing duplicates into equivalence classes. We believe these sets will smooth the difficulties we have had in defining a set of orthogonal operations for schema evolution in nested relations.

Perhaps the most important question is whether we can build out a full operationalized query language. Could that even be extended into a general purpose functional programming language?

7.1 Conclusion

Integrating type/schema operations into a data model allows schema evolution to be handled properly as a first-class concern, solving four of the eight challenge problems identified in prior work. Operational Differencing enables a new kind of version control on data with novel capabilities. It also provides a simplified conceptual model for ad hoc usage. Our preliminary work on operationalizing a query language is intriguing. Overall it would be fair to say that this paper raises more questions than it answers, which we hope is seen as a positive sign of progress and promise.

8 Related Work

OT and CRDT   Operational Differencing is related to Operational Transformation (OT) [7, 29, 51, 56] and Convergent Replicated DataTypes (CRDTs) [59]. Those techniques ensure that multiple replicas distributed across a network will converge to the same state regardless of the order in which operations propagate across the network. To ensure convergence they predetermine how conflicts will be resolved when the operations are first executed. However in version control we need to let humans decide how conflicts will be resolved long after the fact. Version control is about managing divergence not just convergence. On the other hand, most OT and all CRDT algorithms are decentralized, whereas we are centralized: we can synchronize replicas, but only if one of them is designated as a *primary* serializing all transfers.

While we have different requirements it would be fair to say that Operational Differencing is a generalization of OT, and indeed was inspired by it. Our Project and Retract functions can be seen as generalizing OT's Inclusion Transformation

(IT) and Exclusion Transformation (ET). The difference is that our functions return a pair of operations not just one, retraction may fail, and they do not satisfy the correctness properties TP1 and TP2. In a sense we generalize (weaken) OT by removing predetermination of conflict resolution and insertion ordering (making projection non-commutative) and allowing retraction to fail because of dependencies (making it partial). This weakening makes our projection and retraction rules significantly more complex.

There is work on adding branching and conflict resolution to CRDTs [20, 49] and DB replication [19]. CEDA [6] is an OT-based multi-paradigm distributed database. Synql [40] is a CRDT-based approach to replicating relational databases respecting integrity constraints.

Databases   There is a vast literature on schema evolution in databases, however we will limit the discussion here to the minority of operation-based approaches. A fuller survey can be found in Edwards et al. [28].

EvolveDB[25] reverse-engineers the schema into a richer data model and then tracks operations on that model within an IDE. This operation history is used to generate SQL scripts to evolve the database. EdgeDB [41] resolves the ambiguities of state-based schema comparison by asking the developer to clarify what operations were intended, with some answers able to specify custom migration code in a proprietary query language.

Curino et al. [22] initiated a cascade of research on Database Evolution Languages (DELs) using Schema Modification Operators (SMOs) which take "as input a relational schema and the underlying database, and produces as output a (modified) version of the input schema and a migrated version of the database". SMOs can also rewrite queries to accommodate schema changes. Herrmann et al. [34] defined a relationally complete DEL and then extended it into a Bidirectional Database Evolution Language (BiDEL) [35] that has bidirectional transformation capabilities comparable to Baseline, although limited to supporting concurrent schema within one database.

Schema evolution is also a problem for NoSQL [57] databases. While such databases are sometimes called "schemaless" in effect that means the schema is left implicit and tools must try to infer it [60, 61]. Litt et al. [48] uses lenses [31] for bidirectional transformation of JSON. Scherzinger et al. [58] define a set of operators like the relational SMOs discussed above. Chillón et al. [14, 15] offer a comprehensive set of SMOs over a data model unifying SQL and NoSQL that can do query rewriting.

Schema evolution has also been studied for Object Oriented Databases(OODB) [5, 47]. Smalltalk [33] is itself an OODB, persisting all object instances in an *image file* with some evolution capabilities incorporated in the programming environment [33, pp.252-272]. Gemstone turns the Smalltalk image into a production-quality database and accordingly provides a complex schema evolution API [64]. Kamina et al. [43] extend a relational DEL for persistent objects.

Data Science   AI assistants [54] provide a mechanism for interactively correcting transformations inferred by state-based schema comparisons. Wrangler [44] observes interactive manipulation of sampled data to write data transformation programs.

Baseline: Operation-Based Evolution and Versioning of Data

Petersohn et al. [53] present an algebra of operations on Python dataframes that include shape transformations.

Bidirectional Transformations Bidirectional Transformations [24, 39] have been a rich field of research. Coupled Transformations [9, 18, 21] express transformations coordinated between types and their instances by encoding them into functions on Haskell GADTs built with strategy combinators. Visser [66] extends the encoding to transform queries written in point-free style. Lämmel [46] recapitulates coupled transformations within logic programming which extends it to transform logic programs.

Lenses [31, 37] are a linguistic approach to bidirectional transformations that have been the subject of much research. Carvalho et al. [12] use lenses explicitly specified inside code to express its evolution.

Model Driven Engineering Unlike the textual artifacts involved in other domains, models typically assign unique identifiers enabling more precise differencing and mergeing [1]. Models are often themselves modeled by a metamodel, which lifts the evolution problem to the metalevel: models must be migrated through the evolution of their metamodel, analogously to changing the grammar of a programming language. [36]. Cicchetti et al. [17] study metamodel evolution in the context of divergent changes. Vermolen et al. [65] define a DSL for evolving a data model by generating SQL migrations on the backing database. They develop comparable capabilities to Baseline, although non-interactive. They consider the evolution of OO-style subclass hierarchies which goes beyond our and most other work. Storm [62] proposes a research agenda for thoroughly live DSL programming focusing on *Semantic Deltas* that unify edit-time and run-time change. Exelmans et al. [30] offer an operation-based solution to our challenge problem of *Live Editing Domain-Specific Languages* with connections to our approach.

A Deletion and Tombstones

Oster et al. [51] first reported a flaw in how all prior OT algorithms handled deletion. Insertions on either side of a deletion can become conflated and misordered. The now-standard solution is that deletions should leave behind a *tombstone* as a placeholder to keep the insertions on either side separate. The tombstone is hidden from the user, but still occupies space in the string.

The current implementation of Baseline does not leave tombstones in the state. Instead they are tracked in the affected insert operations. Each insert operation includes a sequence of IDs of "virtual tombstones" that record the fact that the insertion was shifted right over a deletion. When an insert operation is projected through a deletion of its insertion point (the before paramater) the insertion point is shifted to the right of the deletion and the ID of the deleted element is appended to the list of tombstones in the insert operation. When inserts at the same point are

projected through each other their tombstone lists are compared to decide which to shift in front of the other. We believe that this is a novel solution to the tombstone problem, however it substantially complicates the projection and retraction rules for inserts and deletes.

State tombstones are easier to understand, so we have adopted them in this paper's description of Baseline. Tombstones have also proven useful in the UI as placeholders when diffing insertions and deletions. We plan to adopt them in the next implementation.

B   Selective Undo

Undo is an essential feature when editing any sort of information, although it is frustrating that the common approach only allows undoing the last operation. Often we only realize mistakes later. *Selective undo* [10] allows the user to undo past operations. Intuitively, undoing a past operation should result in what "would have happened" if that operation had never occurred but all subsequent operations still did.

It is tempting to think of selective undo as simply deleting an operation from the timeline. There are two problems with this view. Firstly, subsequent operations might depend upon the effects of that operation. Subsequent operations have to somehow be scrubbed of all traces of the undone operation. Secondly, in the setting of collaborative replicas or version control we can't just change history—we need to calculate the delta to the current state that will result in the same thing as that hypothetical changed history. In this way the undo operation can be shared with collaborators like any other operation.

Selective undo has been implemented in OT [63, 67] and Baseline takes a similar approach, diagrammed below.

$$\{\} \xrightarrow{\ a_{1\cdots i-1}\ } A_{i-1} \xrightarrow{\quad a_i \quad} A_i \xrightarrow{\ a_{i+1\cdots n}\ } A_n$$
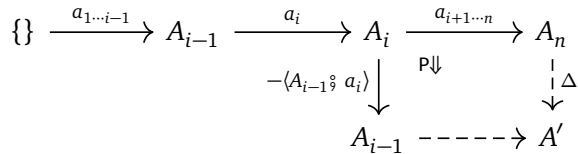
$$\left. {-\langle A_{i-1} \, \mathring{,}\, a_i \rangle} \right\downarrow \qquad \mathsf{P}{\Downarrow} \qquad \vdots\,{\Delta}$$

$$A_{i-1} \dashrightarrow A'$$

The top row of the diagram shows the complete history of A starting at the unit value $\{\}$ and ending with A_n. To undo a past operation a_i we first calculate its *inverse* $-\langle A_{i-1} \, \mathring{,}\, a_i \rangle$ that takes the state A_i back to the state A_{i-1}. In general the inverse takes a timeline and produces a sequence of operations that may depend upon the initial state. For example if a_i is a delete operation its inverse will first re-insert the deleted element at the proper location and then completely restore its value in the prior state.

Given the inverse operations we then project the subsequent timeline through them. Note the downward direction of the projection, which causes subsequent operations to override the inverse. For example if we undo a past write operation but there were subsequent writes to the same location then undoing should have no effect. The projection produces the Δ operations on the right hand side, which is the delta we are looking for. We execute the delta on A producing the result of the undo in A'. The projection also produces the dashed arrow on the bottom, which is not utilized, but is

Figure 11 A selection is a path through Miller columns containing indented outlines and flat tables.

Figure 12 A diff is displayed across a split screen with coordinated selection.

of interest because it is that hypothetical history scrubbed of the effects of the undone operation we imagined earlier.

C Visualizing Structure and Change

A primary goal of Baseline is useability. We believe that people should have the same power as programs, entailing that all state is visualized in a UI and that all operations are manifested as direct manipulations in it. The same goes for our version control capabilities: differences are visualized and can be transferred. There is a playable demo (see Appendix D) and a short video presentation at https://www.hytradboi.com/2025/3b6de0f0-c61c-4e70-9bae-cca5a0e5bb7b-db-usability-as-if.

The following discussion concentrates on the rationale behind the key UI design decisions. Figure 11 displays a screen shot. ① The left hand column shows the history of operations on this database. History sidebars are a common technique but we were particularly inspired by Wrangler [44].

The three columns to the right are Miller columns [68] that display a path drilling into and across nested data structures. Usually Miller columns display a simple list, as in the MacOS Finder column view of directories. We elaborate that convention to show in each column either an indented outline or a flat table. Nested structures are indicated with summaries that when clicked open a new column to the right. Cyan outlines show the selected path through the columns. ② is the top-level structure of this document. ③ is the orders table within it. The user has selected a link to the customers

table, which is opened in the column ④. Selection not only drills into nested structures but also follows links crossing the hierarchy, as if the target table of the link was nested inside it. This design was inspired as a reaction to Ultorg [2, 3] which uses nested tables to represent queries crossing a flat relational database. Ultorg provides a single view of a nested structure whereas our use of Miller columns provides only a view of the context of a path within the structure. The compensation for that restriction is that each column contains a visually simpler and context-independent layout, keeping neighbors at each level adjacent to each other rather than being splayed across lower layers.

Figure 12 displays our representation of diffs. The challenge is that Operational Differencing can do fine-grained comparisons across schema changes. We split the screen in half vertically to compare two databases. Here we are displaying the differences due to database normalization between Figure 8a in the top half and Figure 8c in the bottom half. Differences on each side are given a green background, and the context menu on them provides a command to transfer that difference to the other side.

The hard problem is not showing what has *changed* but showing what is the *same* across complex structural transformations. We use *coordinated selection*: At ① the cyan outline indicates we have selected the customers table cell containing Wile E Coyote. At ② the corresponding cell in the other database has been outlined in white, even though it is in the orders table. We automatically compute the corresponding location of the selection by transfering a noop operation to the other side.

We first tried conventional representations of textual diffs, both side-by-side and interleaved (AKA unified). They have the advantage of keeping corresponding locations close to each other on screen, but only when the differences are limited to inserting and deleting text. They did not adapt well to complex data structures and complex structural transformations. Our split-screen approach handles these cases as well, and has the advantage that the diff view keeps the layout of the data the same as in a normal view.

D The Playable Demo

It is difficult to convey the experience of an interactive user interface in a written narrative, so we have developed a *Playable Demo*. We adopt *onboarding* techniques that commercial software uses to teach new users. A scaffold over the UI walks the user through a demo scenario, stepping through a written narrative that prompts each user action. Users that follow the main quest are given a frustration-free experience going at their own pace. The more adventurous are free to roam at their own risk, but can always respawn where they left the path. The demo can be accessed at https://thebaseline.dev/Prog26submission/. We encourage viewers to share their experience in a survey at the end.

Can a playable demo do a better job explaining new user interface ideas than written narratives and recorded demos? Our early results are mixed.

- Using an open source onboarding library was a mistake. A fully integrated experience requires a fully custom implementation.
- To make a playable demo robust it needs to know exactly what each user step should do and validate it. Specifying that with code is laborious (so much so that we did not do it). Instead we should use the history recorded in a play-through as an oracle. That would also allow "fast forwarding" through the demo, a repeated user request.
- A well-presented demo can lull viewers into suspending disbelief. A playable demo is brutally realistic. Lack of polish is glaringly obvious. Going off script quickly reveals holes in the prototype implementation. These effects make playable demos a risky proposition in competitive academic venues.
- Despite these negatives, users consistently report that when following the script they feel they understand what is being presented.

E Technical Dimensions

We can heuristically evaluate Baseline using the *Technical Dimensions of Programming Systems* [42] framework.

Interaction Strictly speaking Baseline has two **modes of interaction**: user and developer, however user mode is a strict subset of developer mode (no type operations) so it is fair to say there is really only one mode of interaction. Operationalized queries minimize **abstraction construction** since they correspond exactly to interactive actions in the UI, suporting Programming by Demonstration. Operationalizing all change shortens **feedback loops**: The Gulf of Evaluation is reduced because all operations are concrete direct manipulations; The Gulf of Interpretation is reduced by visualizing changes in terms of concrete difference operations. Version control of data enables collaborative feedback loops that are not practical with conventional techniques.

Notation Baseline currently has a **primary notation** that is both a **surface and internal notation**: sequences of operations. However we expect general programming language features will use a superset **overlapping notation** that evaluates into the base one. Arguably because Baseline involves a rich set of operations they do not form a **uniform notation** and generate a complex **expression geography**.

Conceptual structure Baseline is attempting to build a common mechanism for all dimensions of change management, in order to maximize **conceptual integrity** even at the cost of **openness**. Our approach to **commonality** rejects the dichotomy between static/explicit and dynamic/implicit structure: all structure is explicit but can be dynamically refactored as needed. **Composability** is uniformly through operation sequencing.

Customizability   Operations are executed interactively without **staging**, but operations cen be staged in branches and transfered later when desired. Baseline is not **self-sustaining** because it is not implemented in itself and we do not hold that as a goal. Our ID-annotated tree structures provide some level of **addressing and externalizability**.

Complexity, Errors, and Adoptability   Baseline is still very immature in these dimensions.

References

[1]   Marcus Alanen and Ivan Porres. "Difference and Union of Models". In: *«UML» 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*. Edited by Perdita Stevens, Jon Whittle, and Grady Booch. Volume 2863. Lecture Notes in Computer Science. Springer, 2003, pages 2–17. DOI: 10.1007/978-3-540-45221-8_2.

[2]   Eirik Bakke. *Ultorg*. 2025. URL: https://www.ultorg.com (visited on 2025-09-14).

[3]   Eirik Bakke. "Expressive Query Construction through Direct Manipulation of Nested Relational Results". Available at https://dspace.mit.edu/handle/1721.1/107280. PhD thesis. Massachusetts Institute of Technology, Sept. 2016.

[4]   F. Bancilhon and N. Spyratos. "Update semantics of relational views". In: *ACM Transactions on Database Systems* 6.4 (Dec. 1981), pages 557–575. ISSN: 0362-5915. DOI: 10.1145/319628.319634.

[5]   Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. "Semantics and implementation of schema evolution in object-oriented databases". In: *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. SIGMOD '87. San Francisco, California, USA: Association for Computing Machinery, 1987, pages 311–322. ISBN: 0897912365. DOI: 10.1145/38713.38748.

[6]   David Barrett-Lennard. *Introducing CEDA*. 2010. URL: https://cedanet.com.au/ceda/papers/Introducing%20CEDA.pdf (visited on 2025-09-14).

[7]   David Barrett-Lennard. *Operational Transformation*. 2010. URL: https://cedanet.com.au/ceda/ot/ (visited on 2025-09-14).

[8]   Antranig Basman. "The Naturalist's Friend - A case study and blueprint for pluralist data tools and infrastructure". In: *Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2019, Newcastle University, UK, August 28 - 30, 2019*. Edited by Mariana Marasoiu, Luke Church, and Lindsay Marshall. Psychology of Programming Interest Group, Aug. 2019. URL: https://ppig.org/papers/2019-ppig-30th-basman/.

[9] Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, and Joost Visser. "Coupled schema transformation and data conversion for XML and SQL". In: *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*. PADL'07. Nice, France: Springer-Verlag, 2007, pages 290–304. ISBN: 3540696083. DOI: 10.1007/978-3-540-69611-7_19.

[10] Thomas Berlage. "A selective undo mechanism for graphical user interfaces based on command objects". In: *ACM Trans. Comput.-Hum. Interact.* 1.3 (Sept. 1994), pages 269–294. ISSN: 1073-0516. DOI: 10.1145/196699.196721. URL: https://doi.org/10.1145/196699.196721.

[11] Margaret M. Burnett and Brad A. Myers. "Future of End-User Software Engineering: Beyond the Silos". In: *Future of Software Engineering Proceedings*. FOSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pages 201–211. ISBN: 9781450328654. DOI: 10.1145/2593882.2593896.

[12] Luís Carvalho and João Costa Seco. "A Language-Based Version Control System for Python". In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Edited by Jonathan Aldrich and Guido Salvaneschi. Volume 313. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 9:1–9:27. ISBN: 978-3-95977-341-6. DOI: 10.4230/LIPIcs.ECOOP.2024.9. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.9.

[13] Scott Chacon and Ben Straub. *Pro Git*. 2nd. USA: Apress, 2014. ISBN: 1484200772.

[14] Alberto Hernández Chillón, Meike Klettke, Diego Sevilla Ruiz, and Jesús García Molina. *A Taxonomy of Schema Changes for NoSQL Databases*. 2022. arXiv: 2205.11660 [cs.DB].

[15] Alberto Hernández Chillón, Jesús García Molina, José Ramón Hoyos, and María José Ortín. "Propagating Schema Changes to Code: An Approach Based on a Unified Data Model". In: *CEUR Workshop Proceedings*. Volume 3379. CEUR-WS, 2023.

[16] Luke Church, Emma Söderberg, and Elayabharath Elango. "A case of computational thinking: The subtle effect of hidden dependencies on the user experience of version control." In: *PPIG*. 2014, page 16.

[17] Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque, and Alfonso Pierantonio. "On the concurrent versioning of metamodels and models: challenges and possible solutions". In: *Proceedings of the 2nd International Workshop on Model Comparison in Practice*. IWMCP '11. Zurich, Switzerland: Association for Computing Machinery, 2011, pages 16–25. ISBN: 9781450306683. DOI: 10.1145/2000410.2000414.

[18] Anthony Cleve and Jean-Luc Hainaut. "Co-transformations in Database Applications Evolution". In: *Generative and Transformational Techniques in Software Engineering: International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*. Edited by Ralf Lämmel, João Saraiva, and Joost Visser. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pages 409–421. ISBN: 978-3-540-46235-4. DOI: 10.1007/11877028_17.

[19]   Apache CouchDB Contributors. *CouchDB Replication and Conflict Model*. 2025. URL: https://docs.couchdb.org/en/stable/replication/conflicts.html# (visited on 2025-09-14).

[20]   Automerge Contributors. *Automerge Conflicts*. 2025. URL: https://automerge.org/docs/reference/documents/conflicts/ (visited on 2025-09-14).

[21]   Alcino Cunha, José Nuno Oliveira, and Joost Visser. "Type-Safe Two-Level Data Transformation". In: *FM 2006: Formal Methods*. Edited by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pages 284–299. ISBN: 978-3-540-37216-5.

[22]   Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. "Graceful database schema evolution: the PRISM workbench". In: *Proceedings VLDB Endowment* 1.1 (Aug. 2008), pages 761–772. ISSN: 2150-8097. DOI: 10.14778/1453856.1453939.

[23]   Allen Cypher and Daniel Conrad Halbert. *Watch what I do: Programming by demonstration*. MIT press, 1993.

[24]   Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. "Bidirectional Transformations: A Cross-Discipline Perspective". In: *Theory and Practice of Model Transformations*. Edited by Richard F. Paige. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pages 260–283. ISBN: 978-3-642-02408-5.

[25]   Torben Eckwert, Michael Guckert, and Gabriele Taentzer. "EvolveDB: a tool for model driven schema evolution". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. MODELS '22. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pages 61–65. ISBN: 9781450394673. DOI: 10.1145/3550356.3559095.

[26]   Jonathan Edwards and Tomas Petricek. "Interaction vs. Abstraction: Managed Copy and Paste". In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments*. PAINT 2022. Auckland, New Zealand: Association for Computing Machinery, 2022, pages 11–19. ISBN: 9781450399104. DOI: 10.1145/3563836.3568723. URL: https://doi.org/10.1145/3563836.3568723.

[27]   Jonathan Edwards and Tomas Petricek. *Typed Image-based Programming with Structure Editing*. 2021. arXiv: 2110.08993 [cs.PL]. URL: https://arxiv.org/abs/2110.08993.

[28]   Jonathan Edwards, Tomas Petricek, Tijs van der Storm, and Geoffrey Litt. "Schema Evolution in Interactive Programming Systems". In: *The Art, Science, and Engineering of Programming* 9.1 (Oct. 2024). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2025/9/2. URL: http://dx.doi.org/10.22152/programming-journal.org/2025/9/2.

[29] C. A. Ellis and S. J. Gibbs. "Concurrency Control in Groupware Systems". In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. SIGMOD '89. Portland, Oregon, USA: Association for Computing Machinery, 1989, pages 399–407. ISBN: 0897913175. DOI: 10.1145/67544.66963.

[30] Joeri Exelmans, Ciprian Teodorov, and Hans Vangheluwe. "Operation-based versioning as a foundation for live executable models". In: *Software and Systems Modeling* 24.3 (2025), pages 721–739. DOI: 10.1007/s10270-024-01212-x. URL: https://doi.org/10.1007/s10270-024-01212-x.

[31] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem". In: *ACM Transsctions on Programming Languages and Systems* 29.3 (May 2007), 17–es. ISSN: 0164-0925. DOI: 10.1145/1232420.1232424.

[32] Alexander von Franqué. *Never use git cherry-pick*. [Online; accessed 29-August-2025]. URL: https://www.youtube.com/watch?v=WPCxtFkLa7g.

[33] Adele Goldberg. *SMALLTALK-80: The Interactive Programming Environment*. USA: Addison-Wesley Longman Publishing Co., Inc., 1984. ISBN: 0201113724.

[34] Kai Herrmann, Hannes Voigt, Andreas Behrend, and Wolfgang Lehner. "CoDEL – A Relationally Complete Language for Database Evolution". In: *Advances in Databases and Information Systems*. Edited by Morzy Tadeusz, Patrick Valduriez, and Ladjel Bellatreche. Cham: Springer International Publishing, 2015, pages 63–76. ISBN: 978-3-319-23135-8.

[35] Kai Herrmann, Hannes Voigt, Andreas Behrend, Jonas Rausch, and Wolfgang Lehner. "Living in Parallel Realities: Co-Existing Schema Versions with a Bidirectional Database Evolution Language". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pages 1101–1116. ISBN: 9781450341974. DOI: 10.1145/3035918.3064046.

[36] Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. "An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models". In: *Software Language Engineering*. Edited by Brian Malloy, Steffen Staab, and Mark van den Brand. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pages 163–182. ISBN: 978-3-642-19440-5.

[37] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. "Edit lenses". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: Association for Computing Machinery, 2012, pages 495–508. ISBN: 9781450310833. DOI: 10.1145/2103656.2103715. URL: https://doi.org/10.1145/2103656.2103715.

[38] Joshua Horowitz, Devamardeep Hayatpur, Haijun Xia, and Jeffrey Heer. "Sculpin: Direct-Manipulation Transformation of JSON". In: *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*. UIST '25. Association for Computing Machinery, 2025. ISBN: 9798400720376. DOI: 10.1145/3746059.3747651. URL: https://doi.org/10.1145/3746059.3747651.

[39] Zhenjiang Hu, Makoto Onizuka, and Masatoshi Yoshikawa, editors. *Bidirectional Collaborative Data Management: Collaboration Frameworks for Decentralized Systems*. Springer, 2024. ISBN: 978-981-97-6428-0.

[40] Claudia-Lavinia Ignat, Victorien Elvinger, and Habibatou Ba. "Synql: A CRDT-Based Approach for Replicated Relational Databases with Integrity Constraints". In: *Lecture Notes in Computer Science*. Edited by Rolando Martins. Volume LNCS-14677. Distributed Applications and Interoperable Systems. Groningen, Netherlands: Springer Nature Switzerland, June 2024, pages 18–35. DOI: 10.1007/978-3-031-62638-8_2. URL: https://inria.hal.science/hal-04969158.

[41] EdgeDB Inc. *Schema migrations*. 2024. URL: https://docs.edgedb.com/guides/migrations (visited on 2024-05-01).

[42] Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. "Technical Dimensions of Programming Systems". In: *The Art, Science, and Engineering of Programming* 7.3 (2023). ISSN: 2473-7321. DOI: 10.22152/PROGRAMMING-JOURNAL.ORG/2023/7/13.

[43] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. "Evolution Language Framework for Persistent Objects". In: *The Art, Science, and Engineering of Programming* 10.1 (Feb. 15, 2025). DOI: 10.22152/programming-journal.org/2025/10/12. URL: https://2025.programming-conference.org/.

[44] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. "Wrangler: interactive visual specification of data transformation scripts". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: Association for Computing Machinery, 2011, pages 3363–3372. ISBN: 9781450302289. DOI: 10.1145/1978942.1979444.

[45] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. "A Formal Investigation of Diff3". In: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*. Edited by V. Arvind and Sanjiva Prasad. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 485–496. ISBN: 978-3-540-77050-3.

[46] Ralf Lämmel. "Coupled software transformations revisited". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pages 239–252. ISBN: 9781450344470. DOI: 10.1145/2997364.2997366.

[47] Xue Li. "A Survey of Schema Evolution in Object-Oriented Databases". In: *TOOLS 1999: 31st International Conference on Technology of Object-Oriented Languages and Systems, 22-25 September 1999, Nanjing, China*. IEEE Computer Society, 1999, pages 362–371. DOI: 10.1109/TOOLS.1999.796507.

[48] Geoffrey Litt, Peter van Hardenberg, and Henry Orion. *Project Cambria: Translate your data with lenses*. 2020. URL: https://www.inkandswitch.com/cambria.html (visited on 2020-10-01).

[49] Geoffrey Litt, Paul Sonnentag, Max Schöning, Adam Wiggins, Peter van Hardenberg, and Orion Henry. *Patchwork*. 2024. URL: https://www.inkandswitch.com/patchwork/notebook/ (visited on 2025-09-14).

[50] Randall Munroe. *Git*. [Online; accessed 29-August-2025]. URL: https://xkcd.com/1597/.

[51] Gerald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. "Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems". In: *2006 International Conference on Collaborative Computing: Networking, Applications and Worksharing*. 2006, pages 1–10. DOI: 10.1109/COLCOM.2006.361867.

[52] Santiago Perez De Rosso and Daniel Jackson. "What's wrong with git? a conceptual design analysis". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: Association for Computing Machinery, 2013, pages 37–52. ISBN: 9781450324724. DOI: 10.1145/2509578.2509584. URL: https://doi.org/10.1145/2509578.2509584.

[53] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. "Towards scalable dataframe systems". In: *Procedings VLDB Endowment* 13.12 (July 2020), pages 2033–2046. ISSN: 2150-8097. DOI: 10.14778/3407790.3407807.

[54] Tomas Petricek, Gerrit J. J. van den Burg, Alfredo Nazábal, Taha Ceritli, Ernesto Jiménez-Ruiz, and Christopher K. I. Williams. "AI Assistants: A Framework for Semi-Automated Data Wrangling". In: *IEEE Transactions on Knowledge and Data Engineering* 35.9 (2023), pages 9295–9306. DOI: 10.1109/TKDE.2022.3222538.

[55] Tomas Petricek and Jonathan Edwards. "Denicek: Computational Substrate for Document-Oriented End-User Programming". In: *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*. UIST '25. Association for Computing Machinery, 2025. ISBN: 9798400720376. DOI: 10.1145/3746059.3747646. URL: https://doi.org/10.1145/3746059.3747646.

[56] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. "An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors". In: *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*. CSCW '96. Boston, Massachusetts, USA: Association for Computing Machinery, 1996, pages 288–297. ISBN: 0897917650. DOI: 10.1145/240080.240305.

[57] Pramodkumar J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012. ISBN: 9780133036121.

[58] Stefanie Scherzinger, Meike Klettke, and Uta Störl. *Managing Schema Evolution in NoSQL Data Stores*. 2013. arXiv: 1308.0514 [cs.DB].

[59] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011, page 50. URL: https://inria.hal.science/inria-00555588.

[60] Uta Störl and Meike Klettke. "Darwin: A Data Platform for NoSQL Schema Evolution Management and Data Migration". In: *Proceedings of the Workshops of the EDBT/ICDT 2022 Joint Conference, Edinburgh, UK, March 29, 2022*. Volume 3135. CEUR-WS.org, Mar. 2022. URL: https://ceur-ws.org/Vol-3135/dataplat_short3.pdf.

[61] Uta Störl, Meike Klettke, and Stefanie Scherzinger. "NoSQL Schema Evolution and Data Migration: State-of-the-Art and Opportunities (Tutorial)". In: *Proceedings of the 22nd International Conference on Extending Database Technology (EDBT)*. Apr. 2020. DOI: 10.5441/002/edbt.2020.87.

[62] Tijs van der Storm. "Semantic deltas for live DSL environments". In: *2013 1st International Workshop on Live Programming (LIVE)*. 2013, pages 35–38. DOI: 10.1109/LIVE.2013.6617347.

[63] Chengzheng Sun. "Undo any operation at any time in group editors". In: *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*. CSCW '00. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2000, pages 191–200. ISBN: 1581132220. DOI: 10.1145/358916.358990. URL: https://doi.org/10.1145/358916.358990.

[64] GemTalk Systems. *Gemstone Programmer's Guide: Class versions and Instance Migration*. 2015. URL: https://downloads.gemtalksystems.com/docs/GemStone 64/3.2.x/GS64-ProgGuide-3.2/10-ClassHistory.htm (visited on 2024-05-01).

[65] Sander Daniël Vermolen, Guido Wachsmuth, and Eelco Visser. "Generating database migrations for evolving web applications". In: *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*. GPCE '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pages 83–92. ISBN: 9781450306898. DOI: 10.1145/2047862.2047876.

[66] Joost Visser. "Coupled Transformation of Schemas, Documents, Queries, and Constraints". In: *Electronic Notes Theoretical Computer Science* 200.3 (May 2008), pages 3–23. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.04.090.

[67] Stéphane Weiss, Pascal Urso, and Pascal Molli. *A Flexible Undo Framework for Collaborative Editing*. Research Report RR-6516. INRIA, 2008. URL: https://inria.hal.science/inria-00275754.

[68] Wikipedia contributors. *Miller columns — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Miller_columns&oldid=1305075621. [Online; accessed 12-September-2025]. 2025.

About the authors

Jonathan Edwards is an independent researcher interested in simplifying and democratising programming by collapsing the tech stack. He is known for his Subtext programming language experiments and his blog at alarmingdevelopment.org. He has been a researcher at MIT CSAIL and CDG/HARC. He tweets @jonathoda and can be reached at jonathanmedwards@gmail.com.
https://orcid.org/0000-0003-1958-7967

Tomas Petricek is an assistant professor at Charles University. He is interested in finding easier and more accessible ways of thinking about programming. To do so, he combines technical work on programming systems and tools with research into history and philosophy of science. His work can be found at tomasp.net and he can be reached at tomas@tomasp.net.
https://orcid.org/0000-0002-7242-2208