Language-Integrated Query for Temporal Data

Simon Fowler
University of Glasgow
UK
simon.fowler@glasgow.ac.uk

Vashti Galpin
University of Edinburgh
UK
vashti.galpin@ed.ac.uk

James Cheney
University of Edinburgh
UK
jcheney@inf.ed.ac.uk

Abstract

Modern applications often manage time-varying data. Despite decades of research on temporal databases, which culminated in the addition of temporal data operations into the SQL:2011 standard, temporal data query and manipulation operations are unavailable in most mainstream database management systems, leaving developers with the unenviable task of implementing such functionality from scratch.

In this paper, we extend *language-integrated query* to support writing temporal queries and updates in a uniform host language, with the language performing the required rewriting to emulate temporal capabilities automatically on any standard relational database. We introduce two core languages, λ_{TLINQ} and λ_{VLINQ}, for manipulating transaction time and valid time data respectively, and formalise existing implementation strategies by giving provably correct semantics-preserving translations into a non-temporal core language, λ_{LINQ}. We show how existing work on query normalisation supports a surprisingly simple implementation strategy for *sequenced joins*. We implement our approach in the Links programming language, and describe a non-trivial case study based on curating COVID-19 statistics.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; • **Information systems** → **Structured Query Language**; **Temporal data**.

Keywords: language-integrated query, temporal databases, domain-specific languages, multi-tier programming

ACM Reference Format:
Simon Fowler, Vashti Galpin, and James Cheney. 2022. Language-Integrated Query for Temporal Data. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3564719.3568690

GPCE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9920-3/22/12…$15.00
https://doi.org/10.1145/3564719.3568690

1 Introduction

Most interesting programs access or query data stored persistently, often in a database. Relational database management systems (RDBMSs) are the most popular option and provide a standard domain-specific language, SQL, for querying and modifying the data. Ideally, programmers can express the desired queries or updates declaratively in SQL and leave the database to decide how to answer queries or perform updates efficiently and safely (e.g. in the presence of concurrent accesses), but there are many pitfalls arising from interfacing with SQL from a general-purpose language, leading to the well-known *impedance mismatch* problem [10]. These difficulties range from run-time failures due to the generation of queries as unchecked SQL strings at runtime, to serious security vulnerabilities like SQL injection attacks [32].

Among the most successful approaches to overcome the above challenges, and the approach we build upon in this paper, is *language-integrated query*, exemplified by Microsoft's popular LINQ for .NET [25, 36] and in a number of other language designs such as Links [9, 23] and libraries such as Quill [29]. Within this design space we focus on a family of techniques derived from foundational work by Buneman et al. [3] on the *nested relational calculus* (NRC), a core query language with monadic collection types; work by Wong [37] on rewriting NRC expressions to normal forms that can be translated to SQL, which forms the basis of the approach taken in Links [8, 23] and has been adapted to F# [5].

Many interesting database applications involve data that changes over time. Perhaps inevitably, *temporal data management* [19] has a long history. Temporal databases provide powerful features for querying and modifying data that changes over time, and are particularly suitable for modeling time-varying phenomena, such as enterprise data or evolving scientific knowledge, and supporting auditing and transparency about how the current state of the data was achieved, for example in financial or scientific settings.

To illustrate how temporal databases work and why they are useful, consider the following toy example: a temporal to-do list. A temporal database can be conceptualised abstractly as a *function* mapping each possible time instant (e.g. times of day) to a database state [20]. For efficiency in the common case where most of the data is unchanged most of the time, temporal databases are often represented by augmenting each row with an *interval timestamp* indicating the time

period when the row is present. For technical reasons, *closed-open* intervals $[start, end)$ representing the times $start \leq t < end$ are typically used [34].

In our temporal to-do list, the table at each time instant has fields "task", a string field, and "done", a Boolean field. Additional fields "start" and "end" record the endpoints of the time interval during which each row is to be considered part of the table. An end time of ∞ ("forever") reflects that there is no (currently known) end time and in the absence of other changes, the row is considered present from the start time onwards. For example, the table:

task	done	start	end
Go shopping	**true**	11:00	∞
Cook dinner	**false**	11:00	17:30
Walk the dog	**false**	11:00	∞
Cook dinner	**true**	17:30	∞
Watch TV	**false**	11:00	19:00

represents a temporal table where all four tasks were added at 11:00, with "Go shopping" being complete and the others incomplete; at 17:30 "Cook dinner" was marked "done" from then onwards, and at 19:00 "Watch TV" was removed from the table without being completed. Technically, note that this example interprets the time annotations as *transaction time*, that is, the times indicate when certain data was in the database; there is another dimension, *valid time*, and we will discuss both dimensions in greater detail later on.

The problems of querying and updating temporal databases have been well-studied, leading to the landmark language design TSQL2 [33] extending SQL. However, despite decades of effort, only limited elements of TSQL2 were eventually incorporated into the SQL:2011 standard [22] and these features have not yet been widely adopted. Directly implementing temporal queries in SQL is possible, but painful: a TSQL2-style query or update operation may grow by a factor of 10 or more when translated to plain SQL, which leaves plenty of scope for error, and thus these powerful capabilities remain outside the grasp of non-experts. In this paper we take first steps towards reconciling temporal data management with language-integrated query based on query normalisation. We propose supporting temporal capabilities by translation to ordinary language-integrated query and hypothesise that this approach can make temporal data management safer, easier to use and more accessible to non-experts than the current state of affairs. As an initial test of this hypothesis we present a high-level design, a working implementation, and detail our experience with a nontrivial case study.

Although both language-integrated query and temporal databases are now well-studied topics, we believe that their combination has never been considered before. Doing so has a number of potential benefits, including making the power of well-studied language designs such as TSQL2 more accessible to non-expert programmers, and providing a high-level abstraction that can be implemented efficiently in different ways. Our interest is particularly motivated by the needs of scientific database development, where data versioning for accountability and research integrity are very important needs that are not well-supported by conventional database systems [4]. Temporal data management has the potential to become a "killer app" for language-integrated query, and this paper takes a first but significant step towards this goal.

$$
\begin{array}{lrcl}
\text{Types} & A, B & ::= & C \mid A \to^{E} B \mid \mathsf{Bag}(A) \mid \widetilde{(\ell : A)} \mid \mathsf{Table}(A) \\
\text{Base types} & C & ::= & \mathsf{String} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Time} \\
\text{Effects} & e & ::= & \mathbf{read} \mid \mathbf{write} \\
\text{Effect sets} & E & & \\
\text{Terms} & L, M, N & ::= & x \mid c \mid t \\
 & & \mid & \lambda x.M \mid M\ N \mid \odot\{\overrightarrow{M}\} \\
 & & \mid & \mathbf{if}\ L\ \mathbf{then}\ M\ \mathbf{else}\ N \\
 & & \mid & \wr\int \mid \wr M\int \mid M \uplus N \mid \mathbf{for}\ (x \leftarrow M)\ N \\
 & & \mid & \widetilde{(\ell = M)} \mid M.\ell \mid \mathbf{now} \\
 & & \mid & \mathbf{query}\ M \mid \mathbf{get}\ M \mid \mathbf{insert}\ M\ \mathbf{values}\ N \\
 & & \mid & \mathbf{update}\ (x \Leftarrow L)\ \mathbf{where}\ M\ \mathbf{set}\ \widetilde{(\ell = N)} \\
 & & \mid & \mathbf{delete}\ (x \Leftarrow M)\ \mathbf{where}\ N
\end{array}
$$

Figure 1. Syntax of λ_{LINQ}

The overarching contribution of this paper is the first extension of language-integrated query to support transaction time and valid time temporal data.

Concretely, we make three main contributions:

1. Based on λ_{LINQ} (§2), a formalism based on the Nested Relational Calculus (NRC) [31], we introduce typed λ-calculi to model queries and modifications on transaction time (§3) and valid time (§4) databases. We give semantics-preserving translations to λ_{LINQ} for both.
2. We show how existing work on query normalisation allows a surprisingly straightforward implementation strategy for *sequenced joins* (§5).
3. We implement our constructs in the Links functional web programming language, and describe a case study based on curating COVID-19 data (§6).

Although the concepts behind translating temporal queries and updates into non-temporal core languages are well known [34], our core calculi λ_{TLINQ} and λ_{VLINQ} are novel and aid us in showing (to the best of our knowledge) the first correctness results for the translations.

We relegate several details and all proofs to the extended version of the paper [14].

2 Background: Language-Integrated Query

We begin by introducing a basic λ-calculus, called λ_{LINQ}, to model language-integrated query in non-temporal databases. Our calculus is based heavily on the Nested Relational Calculus [31], with support for database modifications heavily inspired by the calculus of Fehrenbach and Cheney [13]. The calculus uses a type-and-effect system to ensure database accesses can occur in 'safe' places, i.e., that we do not attempt to perform a modification operation in the middle of a query. Effects include **read** (denoting a read from a database) and **write** (denoting an update to the database).

Types A, B include base types C, effect-annotated function types $A \to^E B$, unordered collection types $\mathsf{Bag}(A)$, record types $(\ell_i : A_i)_i$ denoting a record with labels ℓ_i and types A_i, and handles $\mathsf{Table}(A)$ for tables containing records of type A. A record is a *base record* if it contains only fields of base type. We assume that the base types includes at least Bool and the Time type, which denotes (abstract) timestamps.

Basic terms include variables x, constants c, table handles t, functions $\lambda x.M$, application $M\ N$, n-ary operations $\odot\{\overrightarrow{M}\}$, and conditionals **if** L **then** M **else** N. We assume that the set of operations contains the usual unary and binary relation operators, as well as the n-ary operations **greatest** and **least** on timestamps which return their largest and smallest arguments respectively. We assume that the set of constants contains timestamps ι of type Time, and two distinguished timestamps $-\infty$ and ∞ of type Time, which denote the minimum and maximum timestamps respectively. The calculus also includes the empty multiset constructor $\wr\,\int$; the singleton multiset constructor $\wr M \int$; multiset union $M \uplus N$; and comprehensions **for** $(x \leftarrow M)\ N$. We write $\wr M_1, \ldots, M_n \int$ as sugar for $\wr M_1 \int \uplus \ldots \uplus \wr M_n \int$. We also have records $(\ell_i = M_i)_i$ and projection $M.\ell$. Term **now** retrieves the current time.

We write **let** $x = M$ **in** N as the usual syntactic sugar for $(\lambda x.N)\ M$, and $M; N$ as sugar for $(\lambda x.N)M$ for some fresh x. We also define whereMN as sugar for **if** M **then** N **else** $\wr\,\int$. We denote unordered collections with a tilde (e.g., $\widetilde{M}$), and ordered sequences with an arrow (e.g., $\overrightarrow{M}$).

The **get** M term retrieves the contents of a table into a bag; **insert** M **values** N inserts values N into table M; **update** $(x \Leftarrow L)$ **where** M **set** $(\ell_i = N_i)_i$ updates table L, updating the fields ℓ_i to N_i of each record x satisfying predicate M. The **delete** $(x \Leftarrow M)$ **where** N term removes those records x in table M satisfying predicate N.

2.1 Typing Rules

Figure 2 shows the typing rules for λ_{LINQ}; the typing judgement has the shape $\Gamma \vdash M{:}A\,!\,E$, which can be read, "Under type environment Γ, term M has type A and produces effects E". The rules are implicitly parameterised by a database schema Σ mapping table names to types of the form $\mathsf{Bag}((\ell_i : C_i)_i)$. Many rules are similar to those of the simply-typed λ-calculus extended with monadic collection operations [3] and a set-based effect system [24], and such standard rules are relegated to the extended version.

Rule T-Query states that a term **query** M is well-typed if M is of a *query type*: either a base type, a record type whose fields are query types, or a bag whose elements are query types. The term M must also only have **read** effects. These restrictions allow efficient compilation to SQL [6, 8].

Rule T-Get states that **get** M has type $\mathsf{Bag}(A)$ if M has type $\mathsf{Table}(A)$, and produces the **read** effect. Rule T-Table states that a table reference follows the type of the table in the schema. T-Insert types a database insert **insert** M **values** N, requiring M to be a table reference of type $\mathsf{Table}(A)$, and the inserted values N to be a bag of type $\mathsf{Bag}(A)$. T-Update ensures the predicate and update terms are typable under an environment extended with the row type, and ensures that all updated values match the type expected by the row. Rule T-Delete is similar. All subterms used as predicates or used to calculate updated terms must be pure (that is, side-effect free), and all modifications have the **write** effect.

2.2 Semantics

Figure 3 shows the syntax and typing rules of values, and the big-step semantics of λ_{LINQ}. Most rules are standard, and presented in the extended version. Values V, W include functions, constants, tables, fully-evaluated records, and fully-evaluated bags $\wr\widetilde{V}\int$. Unlike the unary bag constructor $\wr M \int$, fully-evaluated bags contain an unordered *collection* of values. All values are pure. We write $\oplus$ for record extension, e.g., $(\ell_1 = M) \oplus (\ell_2 = N) = (\ell_1 = M, \ell_2 = N)$.

Since evaluation is effectful (as database operations can update the database), the evaluation judgement has the shape $M \Downarrow_{\Delta,\iota} (V, \Delta')$; this can be read "term M with current database Δ at time ι evaluates to value V with updated database Δ'". A database is a mapping from table names to bags of base records. To avoid additional complexity, we assume evaluation is atomic and does not update the time; one could straightforwardly update the semantics with a **tick** operation without affecting any results.

We use two further evaluation relations for terms which do not write to the database: $M \Downarrow^\star_\iota V$ states that a pure term M (i.e., a term typable under an empty effect set) evaluates to V. Similarly, $M \Downarrow^\star_{\Delta,\iota} V$ states that a term M which only performs the **read** effect evaluates to V. We omit the definitions, which are similar to the evaluation relation but do not propagate database changes (since no changes can occur).

Rule E-Now returns the current timestamp. Rule E-Query evaluates the body of a query using the read-only evaluation relation. Rule E-Get evaluates its subject to a table reference, and then returns the contents of a table. Rule E-Insert does similar, evaluating the values to insert, and then appending them to the contents of the table. Rule E-Update iterates over a table, updating a record if the predicate matches, and leaving it unmodified if not. Finally, E-Delete deletes those rows satisfying the deletion predicate.

λ_{LINQ} enjoys a standard type soundness property.

Proposition 2.1 (Type soundness). *If $\cdot \vdash M{:}A\,!\,E$ then there exists some V and Δ' such that $M \Downarrow_{\Delta,\iota} (V, \Delta')$ and $\cdot \vdash V{:}A\,!\,\emptyset$.*

More importantly, the type-and-effect system ensures that query and update expressions in λ_{LINQ} can be translated to SQL equivalent, even in the presence of higher-order functions and nested query results [5, 6, 8, 23]. This alternative implementation is equivalent to the semantics given here but usually much more efficient since the database query

Term typing

$$\boxed{\Gamma \vdash M : A \,!\, E}$$

$$\text{T-Query}\ \frac{\Gamma \vdash M : \mathsf{Bag}(A) \,!\, E \qquad A :: \mathsf{QType} \qquad E \subseteq \{\mathbf{read}\}}{\Gamma \vdash \mathbf{query}\ M : \mathsf{Bag}(A) \,!\, E} \qquad \text{T-Now}\ \frac{}{\Gamma \vdash \mathbf{now} : \mathsf{Time} \,!\, \emptyset} \qquad \text{T-Get}\ \frac{\Gamma \vdash M : \mathsf{Table}(A) \,!\, E}{\Gamma \vdash \mathbf{get}\ M : \mathsf{Bag}(A) \,!\, \{\mathbf{read}\} \cup E}$$

$$\text{T-Insert}\ \frac{\Gamma \vdash M : \mathsf{Table}(A) \,!\, E \qquad \Gamma \vdash N : \mathsf{Bag}(A) \,!\, \emptyset}{\Gamma \vdash \mathbf{insert}\ M\ \mathbf{values}\ N : () \,!\, \{\mathbf{write}\} \cup E}$$

$$\text{T-Update}\ \frac{\Gamma \vdash L : \mathsf{Table}(A) \,!\, E \qquad A = (\ell_i : B_i)_{i \in I} \qquad \Gamma, x : A \vdash M : \mathsf{Bool} \,!\, \emptyset \qquad (j \in I \wedge \Gamma, x : A \vdash N_j : B_j \,!\, \emptyset)_{j \in J}}{\Gamma \vdash \mathbf{update}\ (x \Leftarrow L)\ \mathbf{where}\ M\ \mathbf{set}\ (\ell_j = N_j)_{j \in J} : () \,!\, \{\mathbf{write}\} \cup E}$$

$$\text{T-Delete}\ \frac{\Gamma \vdash M : \mathsf{Table}(A) \,!\, E \qquad \Gamma, x : A \vdash N : \mathsf{Bool} \,!\, \emptyset}{\Gamma \vdash \mathbf{delete}\ (x \Leftarrow M)\ \mathbf{where}\ N : () \,!\, \{\mathbf{write}\} \cup E}$$

Query typing

$$\boxed{A :: \mathsf{QType}}$$

$$\frac{}{C :: \mathsf{QType}} \qquad \frac{(A_i :: \mathsf{QType})_i}{(\ell_i : A_i)_i :: \mathsf{QType}} \qquad \frac{A :: \mathsf{QType}}{\mathsf{Bag}(A) :: \mathsf{QType}}$$

Figure 2. Typing rules for λ_{LINQ} (selected)

Syntax of values, operations on values, and value typing

$$\begin{array}{rrcl} \text{Values} & V, W & ::= & \lambda x.M \mid c \mid t \mid (\ell_i = V_i)_i \mid \wr \widetilde{V} \int \\ & & & \wr \widetilde{V} \int \mathbin{\hat{\uplus}} \wr \widetilde{W} \int & \triangleq & \wr \widetilde{V} \cdot \widetilde{W} \int \\ & ((\ell_i = V_i)_{i \in I}\ \mathbf{with}\ (\ell_j = W_j)) & \triangleq & (\ell_i = V_i)_{i \in I \setminus J} \oplus (\ell_j = W_j)_{j \in J} \end{array}$$

$$\text{T-BagV}\ \frac{(\Gamma \vdash V_i : A \,!\, \emptyset)_i}{\Gamma \vdash \wr \widetilde{V} \int : \mathsf{Bag}(A) \,!\, \emptyset}$$

Big-step reduction rules

$$\boxed{M \Downarrow_{\Delta,\iota} (V, \Delta')}$$

$$\text{E-Now}\ \frac{}{\mathbf{now} \Downarrow_{\Delta,\iota} (\iota, \Delta)} \qquad \text{E-Query}\ \frac{M \Downarrow^{\star}_{\Delta,\iota} V}{\mathbf{query}\ M \Downarrow_{\Delta,\iota} (V, \Delta)} \qquad \text{E-Get}\ \frac{M \Downarrow_{\Delta,\iota} (t, \Delta')}{\mathbf{get}\ M \Downarrow_{\Delta,\iota} (\Delta'(t), \Delta')}$$

$$\text{E-Insert}\ \frac{M \Downarrow_{\Delta,\iota} (t, \Delta') \qquad N \Downarrow^{\star}_{\iota} V}{\mathbf{insert}\ M\ \mathbf{values}\ N \Downarrow_{\Delta,\iota} ((), \Delta'[t \mapsto \Delta'(t) \mathbin{\hat{\uplus}} V])}$$

$$\text{E-Update}\ \frac{L \Downarrow_{\Delta,\iota} (t, \Delta_1) \qquad \Delta_2 = \Delta_1[t \mapsto \wr \mathrm{upd}(v) \mid v \in \Delta_1(t) \int] \qquad \mathrm{upd}(v) = \begin{cases} (v\ \mathbf{with}\ \widetilde{\ell = W}) & \text{if } M\{v/x\} \Downarrow^{\star}_{\iota} \mathbf{true} \text{ and } (N_i\{v/x\} \Downarrow^{\star}_{\iota} W_i)_i \\ v & \text{if } M\{v/x\} \Downarrow^{\star}_{\iota} \mathbf{false} \end{cases}}{\mathbf{update}\ (x \Leftarrow L)\ \mathbf{where}\ M\ \mathbf{set}\ (\widetilde{\ell = N}) \Downarrow_{\Delta,\iota} ((), \Delta_2)}$$

$$\text{E-Delete}\ \frac{M \Downarrow_{\Delta,\iota} (t, \Delta_1) \qquad \Delta_2 = \Delta_1[t \mapsto \wr v \in \Delta(t) \mid N\{v/x\} \Downarrow^{\star}_{\iota} \mathbf{false} \int]}{\mathbf{delete}\ (x \Leftarrow M)\ \mathbf{where}\ N \Downarrow_{\Delta,\iota} ((), \Delta_2)}$$

Figure 3. Semantics of λ_{LINQ} (selected)

optimiser can takes advantage of any available integrity constraints or statistics about the data.

3 Transaction Time

The first dimension of time we investigate is *transaction time* [35], which records how the *state of the database* changes over time. The key idea behind transaction time databases is that update operations are *non-destructive*, so we can always view a database as it stood at a particular point in time.

Let us illustrate with the to-do list example from the introduction. The original table is on the left. The table after making some changes is shown on the right.

task	done
Go shopping	**true**
Cook dinner	**false**
Walk the dog	**false**
Watch TV	**false**

task	done
Go shopping	**true**
Cook dinner	**true**
Walk the dog	**false**

However, since updates and deletions in λ_{LINQ} are destructive, we have lost the original data. Instead, let us see how this could be handled by a transaction-time database:

task	done	start	end
Go shopping	**true**	11:00	∞
Cook dinner	**false**	11:00	∞
Walk the dog	**false**	11:00	∞
Watch TV	**false**	11:00	∞

There are several methods by which we can maintain the temporal information in the database: for example we could maintain a tracking log which records each entry, or we could use various *temporal partitioning* strategies [34]. In this paper, we use a *period-stamped* representation, where each record in the database is augmented with fields delimiting the interval when the record was present in the database.

The time period is a closed-open representation, meaning that each row is in the database from (and including) the time stated in the *start* column, and is in the database up to (but not including) the time stated in the *end* column. We also assume that $start < end$ always holds.

Here, our database states that all four tasks were entered into the database at 11:00. However, if we then decide to check off "Cook dinner" at 17:30 and delete "Watch TV" at 19:00, we obtain the table shown in the introduction.

Since timestamps are either ∞ or only refer to the past; users do not modify period stamps directly; and the information in the database grows monotonically, we can reconstruct the state of the database at any given time.

3.1 Calculus

λ_{TLINQ} extends λ_{LINQ} with native support for transaction time operations; instead of performing destructive updates, we adjust the end timestamp of affected rows and, if necessary, insert updated rows. λ_{TLINQ} database entries are therefore of the form $V_1^{[V_2,V_3)}$, where V_1 is the record data and V_2 and V_3 are the start and end timestamps.

Figure 4 shows the syntax, typing rules, and semantics of λ_{TLINQ}; for brevity, we show the main differences to λ_{LINQ}. Period-stamped database rows are represented as triples $data^{[start,end)}$ with type $\mathsf{TransactionTime}(A)$, where *data* has type A (the type of each record), and both *start* and *end* have type Time. A row is currently present in the database if its end value is ∞. We introduce three accessors: **data** M extracts the data record from a transaction-time row; **start** M extracts the start time; and **end** M extracts the end time. The **get** construct has an updated type to show that it returns a bag of $\mathsf{TransactionTime}(A)$ values, rather than the records directly. The typing rules for the other constructs remain the same as in λ_{LINQ}.

The accessor rules ET-Data, ET-Start, and ET-End project the expected component of the transaction-time row. Rule ET-Insert period-stamps each record to begin at the current time, and sets the end time to be ∞. Rule ET-Delete records deletions for current rows satisfying the deletion predicate. Instead of being removed from the database, the end times of the affected rows are set to the current timestamp. Finally, rule ET-Update performs updates for current rows satisfying the update predicate. Instead of changing a record directly, the upd definition generates *two* records: the previous record, closed off at the current timestamp, and the new record with updated values, starting from the current timestamp and with end field ∞. Returning to our running example, define $\mathsf{at}(tbl, time)$ to return all records in *tbl* starting before *time* and ending after *time*. We can then query the database as it stood at 18:00:

$\mathsf{at}(t, time) \triangleq$
  for $(x \leftarrow$ **get** $t)$
    where (**start** $x \le time \wedge time <$ **end** $x)$
    $\wr$**data** $x\int$
$\mathsf{at}(tbl, 18{:}00)$

task	done
Go shopping	**true**
Cook dinner	**true**
Walk the dog	**false**
Watch TV	**false**

Let $\mathsf{current}(t) = \mathsf{at}(t, \infty)$ return the current snapshot of t. We can then query the current snapshot of the database:

$\mathsf{current}(tbl) =$

task	done
Go shopping	**true**
Cook dinner	**true**
Walk the dog	**false**

3.2 Translation

We can implement the native transaction-time semantics for λ_{TLINQ} database operations by translation to λ_{LINQ}. Our translation adapts the SQL implementations of temporal operations by Snodgrass [34] to a language-integrated query setting. We prove correctness relative to the semantics.

λ_{TLINQ} has a native representation of period-stamped data, whereas λ_{LINQ} requires table types to be flat. Consequently, the translations require knowledge of the types of each record. We therefore annotate each λ_{TLINQ} database term with the type of table on which it operates (this can be achieved through a standard type-directed translation pass).

The (omitted) translation of λ_{TLINQ} types into λ_{LINQ} types is straightforward, save for $\mathsf{TransactionTime}(A)$ which is translated into a record $(data{:}[\![A]\!], start{:}\mathsf{Time}, end{:}\mathsf{Time})$. The same is true for the basic λ-calculus terms. Timestamped rows $V_{data}^{[V_{start},V_{end})}$ are translated to fit the above record type; specifically, $(data = [\![V_{data}]\!], start = [\![V_{start}]\!], end = [\![V_{end}]\!])$.

Remark 3.1. We have chosen to represent a λ_{TLINQ} period-stamped record as a nested record in λ_{LINQ}, but we could equally adopt a flat representation. Since we build on the Nested Relational Calculus, we take advantage of the ability to return nested results; previous work on query shredding [6] allows us to flatten nested results in a later translation pass. A nested representation is more convenient for our implementation and makes the translation simpler and more compositional, so we mirror this choice in the formalism.

We define the *flattening* of a λ_{TLINQ} row and database as:

$$\downarrow((\ell_i = V_i)_i^{[W_1,W_2)}) \triangleq (\ell_i = V_i)_i \oplus (start = W_1, end = W_2)$$
$$\downarrow\Delta \triangleq [t \mapsto \wr\widetilde{\downarrow D}\int \mid t \mapsto \wr\widetilde{D}\int \in \Delta]$$

Figure 5 shows the translation of λ_{TLINQ} terms into λ_{LINQ}. The translation makes use of three auxiliary definitions. Eta-expansion $\eta(x, \widetilde{\ell})$ eta-expands a variable of record type with respect to a sequence of labels, and $\mathsf{restrict}(x, \widetilde{\ell}, M)$ applies an eta-expanded record to a M under a λ-binder for x (required since the λ_{TLINQ} predicates expect just the data from the record, and not the period-stamping fields). Finally, $\mathsf{isCurrent}(M)$ tests whether the *end* field of M is ∞.

Since timestamped rows are translated to records, the temporal accessor functions are translated to record projection. The **get** function is translated to retrieve the flattened λ_{LINQ} representation of the table and pack it via η-expansion into a bag of nested records. The translation of **insert** extends the provided values with a *start* field referring to the current timestamp and an *end* field set to ∞, before inserting them into the database.

Additional Syntax for λ_{TLINQ}

$$\begin{array}{llll} \text{Types} & A, B & ::= & \cdots \mid \mathsf{TransactionTime}(A) \\ \text{Terms} & L, M, N & ::= & \cdots \mid \mathbf{data}\ M \mid \mathbf{start}\ M \mid \mathbf{end}\ M \end{array} \qquad \begin{array}{llll} \text{Timestamped rows} & D & ::= & V_1^{[V_2, V_3)} \\ \text{Values} & V, W & ::= & \cdots \mid D \end{array}$$

Modified Typing Rules for λ_{TLINQ} $\boxed{\Gamma \vdash V : A \,!\, E}$ $\boxed{\Gamma \vdash M : A \,!\, E}$

T-Row

$$\frac{\Gamma \vdash V_1 : A \,!\, \emptyset \qquad \Gamma \vdash V_2 : \mathsf{Time} \,!\, \emptyset \qquad \Gamma \vdash V_3 : \mathsf{Time} \,!\, \emptyset}{\Gamma \vdash V_1^{[V_2, V_3)} : \mathsf{TransactionTime}(A) \,!\, \emptyset}$$

T-Data

$$\frac{\Gamma \vdash M : \mathsf{TransactionTime}(A) \,!\, E}{\Gamma \vdash \mathbf{data}\ M : A \,!\, E}$$

T-Start

$$\frac{\Gamma \vdash M : \mathsf{TransactionTime}(A) \,!\, E}{\Gamma \vdash \mathbf{start}\ M : \mathsf{Time} \,!\, E}$$

T-End

$$\frac{\Gamma \vdash M : \mathsf{TransactionTime}(A) \,!\, E}{\Gamma \vdash \mathbf{end}\ M : \mathsf{Time} \,!\, E}$$

T-Get

$$\frac{\Gamma \vdash M : \mathsf{Table}(A) \,!\, E}{\Gamma \vdash \mathbf{get}\ M : \mathsf{Bag}(\mathsf{TransactionTime}(A)) \,!\, \{\mathbf{read}\} \cup E}$$

Semantics for λ_{TLINQ} database operations $\boxed{M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (V, \Delta')}$

ET-Data

$$\frac{M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (V_1^{[V_2,V_3)}, \Delta')}{\mathbf{data}\ M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (V_1, \Delta')}$$

ET-Start

$$\frac{M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (V_1^{[V_2,V_3)}, \Delta')}{\mathbf{start}\ M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (V_2, \Delta')}$$

ET-End

$$\frac{M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (V_1^{[V_2,V_3)}, \Delta')}{\mathbf{end}\ M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (V_3, \Delta')}$$

ET-Insert

$$\frac{M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (t, \Delta_1) \qquad N \Downarrow^{\star}_{\iota} \wr \widetilde{V} \wr \qquad vs = \wr v^{[\iota,\infty)} \mid v \in \widetilde{V} \wr \qquad \Delta_2 = \Delta'[t \mapsto \Delta_1(t) \mathbin{\hat{\Uparrow}} vs]}{\mathbf{insert}\ M\ \mathbf{values}\ N \Downarrow^{\mathsf{T}}_{\Delta,\iota} ((), \Delta_2)}$$

ET-Delete

$$\frac{\begin{array}{l} M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (t, \Delta_1) \qquad \Delta_2 = \Delta_1[t \mapsto \wr \mathsf{del}(v) \mid v \in \Delta_1(t) \wr] \\ \mathsf{del}(data^{[start,end)}) = \\ \quad \begin{cases} data^{[start,\iota)} & \text{if } end = \infty \text{ and } N\{data/x\} \Downarrow^{\star}_{\iota} \mathbf{true} \\ data^{[start,end)} & \text{otherwise} \end{cases} \end{array}}{\mathbf{delete}\ (x \Leftarrow M)\ \mathbf{where}\ N \Downarrow^{\mathsf{T}}_{\Delta,\iota} ((), \Delta_2)}$$

ET-Update

$$\frac{\begin{array}{l} L \Downarrow^{\mathsf{T}}_{\Delta,\iota} (t, \Delta_1) \qquad \Delta_2 = \Delta_1[t \mapsto \hat{\Uparrow} \wr \mathsf{upd}(v) \mid v \in \Delta_1(t) \wr] \\ \mathsf{upd}(data^{[start,end)}) = \\ \quad \begin{cases} \wr data^{[start,\iota)}, (data\ \mathbf{with}\ \widetilde{\ell = V})^{[\iota,\infty)} \wr \\ \quad \text{if } M\{data/x\} \Downarrow^{\star}_{\iota} \mathbf{true}, (N_i\{data/x\} \Downarrow^{\star}_{\iota} V_i)_i, \text{and } end = \infty \\ \wr data^{[start,end)} \wr & \text{otherwise} \end{cases} \end{array}}{\mathbf{update}\ (x \Leftarrow L)\ \mathbf{where}\ M\ \mathbf{set}\ (\widetilde{\ell = N}) \Downarrow^{\mathsf{T}}_{\Delta,\iota} ((), \Delta_2)}$$

Figure 4. Syntax, typing rules, and semantics of λ_{TLINQ}

A **delete** is translated as a λ_{LINQ} **update** operation, which sets the *end* record of each affected row to the current timestamp. An **update** is translated in three steps: querying the database to obtain the affected current records, with updated values and timestamps; updating the database to close off the existing affected rows; and materialising the insertion.

3.3 Metatheory

We restrict our attention to *well formed* rows and databases, where the *start* timestamp is less than the *end* timestamp.

Definition 3.1 (Well formed rows and databases)**.** A database Δ is well formed, written $\mathsf{wf}(\Delta)$, if for each timestamped row $V_{data}^{[V_{start}, V_{end})}$ in Δ we have that $V_{start} < V_{end}$.

Definition 3.2 (Maximum timestamp)**.** The *maximum timestamp* of a collection of records $\widetilde{D}$ is defined as the maximum timestamp in the set $\{V_{end} \mid V_{data}^{[V_{start}, V_{end})} \in \widetilde{D}, V_{end} \neq \infty\}$, or $-\infty$ if the set is empty.

The maximum timestamp of a database Δ, written $\mathsf{max}(\Delta)$, is the maximum timestamp of all its constituent tables.

Again, λ_{TLINQ} enjoys type soundness.

Proposition 3.2 (Type soundness (λ_{TLINQ}))**.** *If $\cdot \vdash M : A \,!\, E$, then given a $wf(\Delta)$ and ι such that $max(\Delta) \leq \iota$, then there exists some V and well formed Δ' such that $M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (V, \Delta')$.*

We can now show that the translation is correct:

Theorem 3.1. If $\cdot \vdash M : A \,!\, E$ and $M \Downarrow^{\mathsf{T}}_{\Delta,\iota} (V, \Delta')$ where $\mathsf{wf}(\Delta)$ and $\mathsf{max}(\Delta) \leq \iota$, then $[\![M]\!] \Downarrow_{\downarrow\Delta,\iota} ([\![V]\!], \downarrow\Delta')$.

4 Valid Time

The other dimension of time we will look at is *valid time*, which tracks when something is true in the *domain being modelled*. Each timestamp therefore defines the *period of validity* (PV) of each record.

Unlike in a transaction time database, the database does not necessarily grow monotonically since we can apply destructive updates and deletions. Furthermore, whereas in a transaction time database timestamps can only refer to the past (or ∞), in a valid time database we may state that a row is valid until some specific point in the future (for example, the end of a fixed-term employment contract). A further difference from transaction time databases is that users *can* modify timestamps directly, and can also apply updates and deletions over a time period. Let us illustrate with the 'employees' table of an HR database:

name	position	salary	start	end
Alice	Lecturer	40000	2010	2018
Alice	Senior Lecturer	50000	2018	∞
Bob	PhD Student	15000	2019	2023
Charles	PhD Student	15000	2018	2022

Auxiliary Definitions

$$\eta(x, \widetilde{\ell}) \triangleq (\ell_i = x.\ell_i)_i$$
$$\text{restrict}(x, \widetilde{\ell}, M) \triangleq (\lambda x.M)\ \eta(\widetilde{\ell}, x)$$
$$\text{isCurrent}(M) \triangleq M.end = \infty$$

Translation on database terms

$[\![\textbf{data}\ M]\!] = [\![M]\!].data$

$[\![\textbf{start}\ M]\!] = [\![M]\!].start$

$[\![\textbf{end}\ M]\!] = [\![M]\!].end$

$[\![\textbf{get}^{(\ell_i:A_i)_i}\ M]\!] =$
  query
    for $(x \leftarrow \textbf{get}\ [\![M]\!])$
      $\wr(data = \eta(x, \widetilde{\ell}), start = x.start, end = x.end)\int$

$[\![\textbf{insert}^{(\ell_i:A_i)_i}\ M\ \textbf{values}\ N]\!] =$
  let *rows* =
    for $(x \leftarrow [\![N]\!])$
      $\wr\eta(x, \widetilde{\ell}) \oplus (start = \textbf{now}, end = \infty)\int$
  in
  insert $[\![M]\!]$ **values** *rows*

$[\![\textbf{delete}^{(\ell_i:A_i)_i}\ (x \Leftarrow M)\ \textbf{where}\ N]\!] =$
  update $(x \Leftarrow [\![M]\!])$
    where $(\text{restrict}(x, \widetilde{\ell}, [\![N]\!]) \wedge \text{isCurrent}(x))$
    set $(end = \textbf{now})$

$[\![\textbf{update}^{(\ell_i:A_i)_{i\in I}}\ (x \Leftarrow L)\ \textbf{where}\ M\ \textbf{set}\ (\ell = N_j)_{j\in J}]\!] =$
  let $tbl = [\![L]\!]$ **in**
  let *affected* =
    query
      for $(x \leftarrow \textbf{get}\ tbl)$
        where $((\text{restrict}(x, \{\ell_i\}_{i\in I}, [\![M]\!]) \wedge \text{isCurrent}(x)))$
        $\left\wr \begin{array}{l} (\ell_i = x.\ell_i)_{i\in I\setminus J} \oplus \\ (\ell_j = \text{restrict}(x, \{\ell_i\}_{i\in I}, [\![N_j]\!]))_{j\in J} \oplus \\ (start = \textbf{now}, end = \infty) \end{array} \right\int$
  in
  update $(x \Leftarrow tbl)$
    where $(\text{restrict}(x, \widetilde{\ell}, [\![M]\!]) \wedge \text{isCurrent}(x))$
    set $(end = \textbf{now})$;
  insert *tbl* **values** *affected*

Figure 5. Translation from λ_{TLINQ} into λ_{LINQ}

The first modification is to hire Dolores as a professor, on an open-ended contract. As this is an insertion operation on the database at the current moment in time, it is known as a *current insertion*. We can write the following query:

insert *employees* **values**
  (*name* = "Dolores", *position* = "Professor", *salary* = 70000)

Next, we want to record that Alice has resigned. We can write the following *current deletion* query:

delete $(x \Leftarrow employees)$ **where** $x.name$ = "Alice"

The resulting table state shows that Dolores is a Professor from the current time onwards, and that the 'end' field of Alice's current row is updated to the current year:

name	position	salary	start	end
Alice	Lecturer	40000	2010	2018
Alice	Senior Lecturer	50000	2018	2022
Dolores	Professor	70000	2022	∞
...	...	...	...	...

A powerful feature of valid-time databases is the ability to perform *sequenced modifications*, which apply an update or deletion over a particular *period of applicability* (PA). In fact, current modifications are a special case of sequenced modifications applied from **now** until ∞. Suppose that Dolores has agreed to act as Head of School between 2023 and 2028. We can record this using a *sequenced update* query:

update sequenced $(x \Leftarrow employees)$
  between 2023 **and** 2028 **where** $(x.name$ = "Dolores")
  set (*position* = "Head of School")

with the resulting table being:

name	position	salary	start	end
Dolores	Professor	70000	2022	2023
Dolores	Head of School	70000	2023	2028
Dolores	Professor	70000	2028	∞
...	...	...	...	...

Since the period of applicability of the sequenced update is entirely contained within the period of validity of Dolores's row, we end up with three rows: the unchanged record before and after the PA, and the updated record during the PA. We also allow a *sequenced deletion*, and a *sequenced insertion*, where each record's period of validity is given explicitly.

Additionally, suppose that all PhD students are to be given a 1-year extension due to the disruption caused by the pandemic; in this case we want to change the period of validity directly. This is known as a *nonsequenced update*. We cannot express this modification using either current or sequenced modifications since we must calculate the each row's new end date from its previous end date. We can write the modification as follows, noting that we can both read from, and write to, the period of validity directly:

update nonsequenced $(x \Leftarrow employees)$
  where $((\textbf{data}\ x).position$ = "PhD student")
  set $()$ **valid from** $(\textbf{start}\ x)$ **to** $(\textbf{end}\ x + 1)$

The resulting table shows that the 'end' field of Bob's and Charles' records are updated to 2024 and 2023 respectively:

name	position	salary	start	end
Bob	PhD Student	15000	2019	2024
Charles	PhD Student	15000	2018	2023
...	...	...	...	...

4.1 Calculus

The λ_{VLINQ} calculus gives a direct semantics to valid time operations. Like λ_{TLINQ}, λ_{VLINQ} has a native notion of a period-stamped database row, with accessors for the data and each timestamp; the typing rules, reduction rules, and translations are straightforward adaptations of those in λ_{TLINQ}.

Figure 6 shows how the syntax and typing rules for λ_{VLINQ} differ from those of λ_{LINQ}. Unlike in λ_{TLINQ}, we can use the term $M_1^{[M_2,M_3)}$ to construct a valid-time row. Sequenced insertions are described by the term **insert sequenced** M **values** N where TV-SeqInsert ensures that N is a bag of timestamped records. Sequenced updates are described by:

Syntax

$$
\begin{array}{llll}
\text{Types} & A, B & ::= & \mathsf{ValidTime}(A) \\
\text{Terms} & L, M, N & ::= & \cdots \mid L^{[M,N)} \mid \textbf{data}\, M \mid \textbf{start}\, M \mid \textbf{end}\, M \mid \textbf{insert sequenced}\ M\ \textbf{values}\ N \\
& & \mid & \textbf{update sequenced}\ (x \Leftarrow L)\ \textbf{between}\ M_1\ \textbf{and}\ M_2\ \textbf{where}\ M_3\ \textbf{set}\ (\widetilde{\ell = N}) \\
& & \mid & \textbf{update nonsequenced}\ (x \Leftarrow L)\ \textbf{where}\ M\ \textbf{set}\ (\widetilde{\ell = N})\ \textbf{valid from}\ N_1'\ \textbf{to}\ N_2' \\
& & \mid & \textbf{delete sequenced}\ (x \Leftarrow L)\ \textbf{between}\ M_1\ \textbf{and}\ M_2\ \textbf{where}\ N \\
& & \mid & \textbf{delete nonsequenced}\ (x \Leftarrow M)\ \textbf{where}\ N \\
\text{Values} & V, W & ::= & \cdots \mid V_1^{[V_2,V_3)}
\end{array}
$$

Typing rules $\boxed{\Gamma \vdash M : A \,!\, E}$

$$\textsc{TV-Get}\ \frac{\Gamma \vdash M{:}\mathsf{Table}(A) \,!\, E}{\Gamma \vdash \textbf{get}\ M{:}\mathsf{Bag}(\mathsf{ValidTime}(A)) \,!\, \{\textbf{read}\} \cup E} \qquad \textsc{TV-SeqInsert}\ \frac{\Gamma \vdash M{:}\mathsf{Table}(A) \,!\, E \qquad \Gamma \vdash N{:}\mathsf{Bag}(\mathsf{ValidTime}(A)) \,!\, \emptyset}{\Gamma \vdash \textbf{insert sequenced}\ M\ \textbf{values}\ N{:}() \,!\, \{\textbf{write}\} \cup E}$$

$$\textsc{TV-SeqUpdate}\ \frac{\Gamma \vdash L{:}\mathsf{Table}(A) \,!\, E \qquad A = (\ell_i : B_i)_{i \in I} \qquad \Gamma \vdash M_1{:}\mathsf{Time} \,!\, \emptyset \qquad \Gamma \vdash M_2{:}\mathsf{Time} \,!\, \emptyset \qquad \Gamma, x : A \vdash M_3{:}\mathsf{Bool} \,!\, \emptyset \qquad (j \in I \wedge \Gamma, x : A \vdash N_j{:}B_j \,!\, \emptyset)_{j \in J}}{\Gamma \vdash \textbf{update sequenced}\ (x \Leftarrow L)\ \textbf{between}\ M_1\ \textbf{and}\ M_2\ \textbf{where}\ M_3\ \textbf{set}\ (\ell_j = N_j)_{j \in J}{:}() \,!\, \{\textbf{write}\} \cup E}$$

$$\textsc{TV-NonseqUpdate}\ \frac{\begin{array}{c}\Gamma \vdash L{:}\mathsf{Table}(A) \,!\, E \qquad A = (\ell_i : B_i)_{i \in I} \qquad \Gamma, x : \mathsf{ValidTime}(A) \vdash M{:}\mathsf{Bool} \,!\, \emptyset \\ (j \in I \wedge \Gamma, x : \mathsf{ValidTime}(A) \vdash N_j{:}B_j \,!\, \emptyset)_{j \in J} \qquad \Gamma, x : \mathsf{ValidTime}(A) \vdash N_1'{:}\mathsf{Time} \,!\, \emptyset \qquad \Gamma, x : \mathsf{ValidTime}(A) \vdash N_2'{:}\mathsf{Time} \,!\, \emptyset\end{array}}{\Gamma \vdash \textbf{update nonsequenced}\ (x \Leftarrow L)\ \textbf{where}\ M\ \textbf{set}\ (\ell_j{=}N_j)_{j \in J}\ \textbf{valid from}\ N_1'\ \textbf{to}\ N_2'{:}() \,!\, \{\textbf{write}\} \cup E}$$

$$\textsc{TV-SeqDelete}\ \frac{\Gamma \vdash L{:}\mathsf{Table}(A) \,!\, E_1 \qquad \Gamma \vdash M_1{:}\mathsf{Time} \,!\, E_2 \qquad \Gamma \vdash M_2{:}\mathsf{Time} \,!\, E_3 \qquad \Gamma, x{:}A \vdash N{:}\mathsf{Bool} \,!\, \emptyset}{\Gamma \vdash \textbf{delete sequenced}\ (x \Leftarrow L)\ \textbf{between}\ M_1\ \textbf{and}\ M_2\ \textbf{where}\ N{:}() \,!\, \{\textbf{write}\} \cup E_1 \cup E_2 \cup E_3}$$

$$\textsc{TV-NonseqDelete}\ \frac{\Gamma \vdash M{:}\mathsf{Table}(A) \,!\, E \qquad \Gamma, x : \mathsf{ValidTime}(A) \vdash N{:}\mathsf{Bool} \,!\, \emptyset}{\Gamma \vdash \textbf{delete nonsequenced}\ (x \Leftarrow M)\ \textbf{where}\ N{:}() \,!\, \{\textbf{write}\} \cup E}$$

Figure 6. Syntax and typing rules for λ_{VLINQ}

update sequenced $(x \Leftarrow L)$ **between** M_1 **and** M_2 **where** M_3 **set** $(\widetilde{\ell = N})$

Terms M_1 and M_2 must be of type Time, referring to the period of applicability of the sequenced update. Nonsequenced updates are described by the term:

update nonsequenced $(x \Leftarrow L)$
where M **set** $(\widetilde{\ell = N})$ **valid from** N_1' **to** N_2'

with TV-NonseqUpdate stating that the database row (including period information) is bound as x in the predicate M, update terms N_j, and new time periods N_1' and N_2'.

Finally, the term:

delete sequenced $(x \Leftarrow L)$ **between** M_1 **and** M_2 **where** N

describes a sequenced deletion which removes the portion of each record satisfying N between times M_1 and M_2.

Since current insertions, updates, and deletions are special cases of sequenced operations, we need not consider them explicitly; for completeness, direct semantics can be found in the extended version. Instead, we show macro translations to the sequenced constructs. Current insertions can be implemented by desugaring to sequenced insertions, annotating each row with $[\textbf{now}, \infty)$:

$$\textbf{insert}\ M\ \textbf{values}\ N \rightsquigarrow \begin{array}{l}\textbf{let}\ \mathit{rows} = \textbf{for}\ (x \leftarrow N) \wr x^{[\mathbf{now},\infty)} \int\ \textbf{in} \\ \textbf{insert sequenced}\ M\ \textbf{values}\ \mathit{rows}\end{array}$$

Current updates and deletions can be implemented as sequenced updates and deletions where the period of applicability spans from **now** until ∞:

$$\begin{array}{l}\textbf{update}\ (x \Leftarrow L)\ \textbf{where}\ M\ \textbf{set}\ (\ell_i = N_i)_i \rightsquigarrow \\ \quad \textbf{update sequenced}\ (x \Leftarrow L) \\ \quad\quad \textbf{between now and}\ \infty\ \textbf{where}\ M\ \textbf{set}\ (\ell_i = N_i)_i\end{array}$$

$$\begin{array}{l}\textbf{delete}\ (x \Leftarrow M)\ \textbf{where}\ N \rightsquigarrow \\ \quad \textbf{delete sequenced}\ (x \Leftarrow M)\ \textbf{between now and}\ \infty\ \textbf{where}\ N\end{array}$$

Fig. 7 shows selected reduction rules: we show sequenced inserts and updates, and nonsequenced updates; the rules for other cases employ similar ideas and are included in the extended version. Nonsequenced updates and deletes are similar to their analogues in λ_{LINQ} but allow access to, and modification of, row timestamps. For sequenced insertions, EV-SeqInsert checks that the period of validity for each row is correct (i.e., that the *start* field is less than the *end* field) and appends the provided bag to the table. Sequenced updates and deletions must account for the various ways that the period of applicability can overlap the period of validity. There are five main cases, corresponding to the five ways two closed-open intervals can overlap (or fail to do so):

Reduction rules $\boxed{M \Downarrow^{\mathsf{V}}_{\Delta,\iota} (V, \Delta')}$

EV-Row

$$\frac{M_1 \Downarrow^{\mathsf{V}}_{\Delta,\iota} (V_1, \Delta_1) \quad M_2 \Downarrow^{\mathsf{V}}_{\iota,\Delta_1} (V_2, \Delta_2) \quad M_3 \Downarrow^{\mathsf{V}}_{\iota,\Delta_2} (V_3, \Delta_3)}{M_1^{[M_2,M_3)} \Downarrow^{\mathsf{V}}_{\Delta,\iota} (V_1^{[V_2,V_3)}, \Delta_3)}$$

EV-SeqInsert

$$\frac{M \Downarrow^{\mathsf{V}}_{\Delta,\iota} (t, \Delta_1) \quad N \Downarrow^{\star}_{\iota} \wr \widetilde{V} \int \quad \forall data^{[start,end)} \in \widetilde{V}.start < end \quad \Delta_2 = \Delta_1[t \mapsto \Delta_1(t) \hat{\uplus} \wr \widetilde{V} \int]}{\textbf{insert sequenced } M \textbf{ values } N \Downarrow^{\mathsf{V}}_{\Delta,\iota} ((), \Delta_2)}$$

EV-SeqUpdate

$$\frac{L \Downarrow^{\mathsf{V}}_{\Delta,\iota} (t, \Delta_1) \quad M_1 \Downarrow^{\star}_{\iota} V_{start} \quad M_2 \Downarrow^{\star}_{\iota} V_{end} \quad V_{start} < V_{end} \quad \Delta_2 = \Delta_1[t \mapsto \left\lfloor \hat{+} \right\rfloor \wr \operatorname{upd}(d) \mid d \in \Delta_1(t) \int]}{\textbf{update sequenced } (x \Leftarrow L) \textbf{ between } M_1 \textbf{ and } M_2 \textbf{ where } M_3 \textbf{ set } (\ell_i = N_i)_i \Downarrow_{\Delta,\iota} ((), \Delta_2)}$$

$$\operatorname{upd}(v^{[start,end)}) = \begin{cases} \wr W^{[start,end)} \int & \text{if } M_3\{v/x\} \Downarrow^{\star}_{\iota} \textbf{true} \text{ and } V_{start} \le start \text{ and } V_{end} \ge end \quad \textbf{(Case 1)} \\ \wr W^{[start,V_{end})}, v^{[V_{end},end)} \int & \text{if } M_3\{v/x\} \Downarrow^{\star}_{\iota} \textbf{true} \text{ and } V_{start} \le start \text{ and } V_{end} < end \quad \textbf{(Case 2)} \\ \wr v^{[start,V_{start})}, W^{[V_{start},V_{end})}, v^{[V_{end},end)} \int & \text{if } M_3\{v/x\} \Downarrow^{\star}_{\iota} \textbf{true} \text{ and } V_{start} > start \text{ and } V_{end} < end \quad \textbf{(Case 3)} \\ \wr v^{[start,V_{start})}, W^{[V_{start},end)} \int & \text{if } M_3\{v/x\} \Downarrow^{\star}_{\iota} \textbf{true} \text{ and } V_{start} > start \text{ and } V_{end} \ge end \quad \textbf{(Case 4)} \\ \wr v^{[start,end)} \int & \text{otherwise} \quad \textbf{(Case 5)} \end{cases}$$

where for all cases, $W = (v \textbf{ with } \widetilde{\ell = W'})$ given $(N_i \Downarrow^{\star}_{\iota} W'_i)_i$

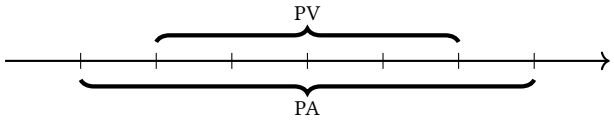
EV-NonseqUpdate

$$\frac{L \Downarrow_{\Delta,\iota} (t, \Delta_1) \quad \Delta_2 = \Delta_1[t \mapsto \wr \operatorname{upd}(d) \mid d \in \Delta_1(t) \int]}{\textbf{update nonsequenced } (x \Leftarrow L) \textbf{ where } M \textbf{ set } (\ell_i = N_i)_i \textbf{ valid from } N'_1 \textbf{ to } N'_2 \Downarrow_{\Delta,\iota} ((), \Delta_2)}$$

$$\operatorname{upd}(D = v^{[start,end)}) = \begin{cases} (v \textbf{ with } \widetilde{\ell = W})^{[W_{start},W_{end})} & \text{if } M\{D/x\} \Downarrow^{\star}_{\iota} \textbf{true} \text{ and } (N_i\{D/x\} \Downarrow^{\star}_{\iota} W_i)_i \text{ and} \\ & \quad N'_1\{D/x\} \Downarrow^{\star}_{\iota} W_{start} \text{ and } N'_2\{D/x\} \Downarrow^{\star}_{\iota} W_{end} \text{ and } W_{start} < W_{end} \\ D & \text{if } M\{D/x\} \Downarrow^{\star}_{\iota} \textbf{false} \end{cases}$$

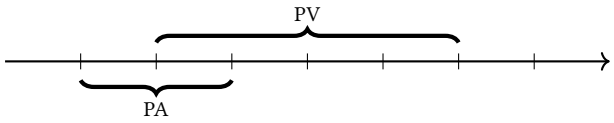
Figure 7. Reduction rules for λ_{VLINQ} (selected)

Case 1: PA overlaps PV entirely



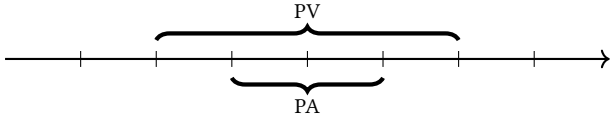
In the case of a sequenced deletion, the entire row will be deleted. In the case of a sequenced update, the entire row will be updated.

Case 2: PA overlaps PV on the left



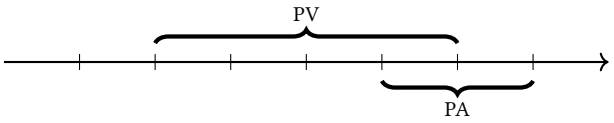
In the case of a sequenced deletion, the overlapping portion will be deleted; in the case of a sequenced update, the overlapping portion will contain the updated values and the remaining portion of the PV will contain the previous values.

Case 3: PA contained within PV



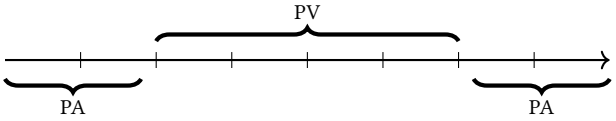
In the case of a sequenced deletion, there will be two records: the portion of the PV before the start of the PA, and the portion of the PV after the end of the PA. In the case of a sequenced update, there will be three records: the portion of the PV before the start of the PA, and the portion of the PV after the end of the PA will contain the original values, and the overlapping portion will contain the updated values.

Case 4: PA overlaps PV on the right



Similar to case 2, but at the end of the PV.

Case 5: PA entirely before or after PV



In the case of either a sequenced deletion or a sequenced update, the row will be unaffected.

4.2 Translation

Figure 8 illustrates the translation from λ_{VLINQ} into λ_{LINQ}. We discuss the translations for sequenced inserts and both sequenced and nonsequenced updates; the other modifications are similar and included in the extended version. As before, we require annotations on each of the database update terms.

Nonsequenced updates and deletions can be updated directly by their corresponding λ_{LINQ} operation; we use an auxiliary definition, lift, which lifts the flat representation into the nested representations expected by the predicate and update fields. Sequenced inserts flatten the contents of the provided bag and map directly to an **insert**.

$$
\begin{array}{l}
(\!|\textbf{insert}^{(\ell_i:A_i)_i}\ \textbf{sequenced}\ M\ \textbf{values}\ N|\!) = \\
\quad \textbf{let}\ tbl = (\!|M|\!)\ \textbf{in} \\
\quad \textbf{let}\ rows = \\
\quad\quad \textbf{for}\ (x \leftarrow (\!|N|\!)) \\
\quad\quad\quad \wr \eta(x.data, \widetilde{\ell}) \oplus (start = x.start, end = x.end) \int \\
\quad \textbf{in} \\
\quad \textbf{insert}\ tbl\ \textbf{values}\ rows \\
\\
\left(\!\!\left| \begin{array}{l} \textbf{update}^{(\ell_i:A_i)_{i\in I}}\ \textbf{sequenced}\ (x \Leftarrow L) \\ \quad \textbf{between}\ M_1\ \textbf{and}\ M_2\ \textbf{where}\ M_3 \\ \quad \textbf{set}\ (\ell_j = N_j)_{j\in J} \end{array} \right|\!\!\right) = \\
\\
\quad\quad \textbf{let}\ tbl = (\!|L|\!)\ \textbf{in} \\
\quad\quad \textbf{let}\ aStart = (\!|M_1|\!)\ \textbf{in} \\
\quad\quad \textbf{let}\ aEnd = (\!|M_2|\!)\ \textbf{in} \\
\quad\quad \textbf{let}\ lRows = \mathsf{startRows}(tbl, \mathsf{pred}, aStart)\ \textbf{in} \\
\quad\quad \textbf{let}\ rRows = \mathsf{endRows}(tbl, \mathsf{pred}, aEnd)\ \textbf{in} \\
\quad\quad \textbf{update}\ (x \Leftarrow tbl) \\
\quad\quad\quad \textbf{where}\ (\mathsf{pred} \wedge (x.start < aEnd) \wedge (x.end > aStart)) \\
\quad\quad\quad\quad \textbf{set} \left(\begin{array}{l} (\ell_j = \mathsf{restrict}(x, \{\ell_i\}_{i\in I}, (\!|N_j|\!)))_{j\in J}, \\ start = \textbf{greatest}(x.start, aStart), \\ end = \textbf{least}(x.end, aEnd) \end{array} \right); \\
\quad\quad \textbf{insert}\ tbl\ \textbf{values}\ lRows; \\
\quad\quad \textbf{insert}\ tbl\ \textbf{values}\ rRows \\
\mathsf{where} \\
\quad \mathsf{pred} \triangleq \mathsf{restrict}(x, \{\ell_i\}_{i\in I}, (\!|M_3|\!)) \\
\quad \mathsf{startRows}(tbl, pred, aStart) \triangleq \textbf{query} \\
\quad\quad \textbf{for}\ (x \leftarrow \textbf{get}\ tbl) \\
\quad\quad\quad \mathsf{where}\ (\mathsf{pred} \wedge (x.start < aStart) \wedge (x.end > aStart)) \\
\quad\quad\quad \wr \eta(x, \{\ell_i\}_{i\in I}) \oplus (start = x.start, end = aStart) \int \\
\quad \mathsf{endRows}(tbl, pred, aEnd) \triangleq (\text{symmetric}) \\
\\
\left(\!\!\left| \begin{array}{l} \textbf{update}^{(\ell_i:A_i)_{i\in I}}\ \textbf{nonsequenced}\ (x \Leftarrow L)\ \textbf{where}\ M \\ \quad \textbf{set}\ (\ell_j = N_j)_{j\in J}\ \textbf{valid from}\ N_1'\ \textbf{to}\ N_2' \end{array} \right|\!\!\right) = \\
\\
\quad \textbf{update}^{(\ell_i:A_i)_{i\in I}}\ (x \Leftarrow (\!|L|\!)) \\
\quad\quad \textbf{where}\ (\mathsf{lift}(x, (\!|M|\!))) \\
\quad\quad\quad \textbf{set} \left(\begin{array}{l} ((\ell_j = \mathsf{lift}(x, (\!|N_j|\!)))_{j\in J}, \\ start = \mathsf{lift}(x, (\!|N_1'|\!)), \\ end = \mathsf{lift}(x, (\!|N_2'|\!))) \end{array} \right) \\
\mathsf{where}\ \mathsf{lift}(x, f) \triangleq \\
\quad (\lambda x.f)\ (data = \eta(x, \{\ell_i\}_{i\in I}), start = x.start, end = x.end)
\end{array}
$$

Figure 8. Translation from λ_{VLINQ} into λ_{LINQ} (selected cases)

The remaining sequenced operations are the most complex to translate. Since a sequenced modification may partition a row, the startRows and endRows functions calculate the records which must be inserted before and after the period of applicability. To translate a sequenced update, we calculate the rows to insert, perform an **update** to set the new values and set the new period of applicability to the overlap between the PA and PV using the **greatest** and **least** functions, and finally materialise the insertions. Sequenced deletions (shown in the extended version) are similar but delete the rows that overlap the PA instead of updating them.

4.3 Metatheory

Evaluation preserves typing and well-formedness.

Proposition 4.1 (Preservation (λ_{VLINQ})). *If* $\cdot \vdash M{:}A\,!\,E$ *and* $M \Downarrow^{V}_{\Delta,\iota} (V, \Delta')$ *for some* $wf(\Delta)$, *then* $\cdot \vdash V{:}A\,!\,\emptyset$ *and* $wf(\Delta')$.

Unlike λ_{LINQ} and λ_{TLINQ}, evaluation in λ_{VLINQ} is *partial* in order to reflect the need for dynamic checks that start times precede end times. In practice, our implementation evaluates temporal updates as single transactions and raises an exception (aborting the transaction) when a well-formedness check fails, but our formalisation assumes updates preserve well-formedness in order to avoid clutter.

Our translation from λ_{VLINQ} into λ_{TLINQ} satisfies the following correctness property:

Theorem 4.1. If $\cdot \vdash M{:}A\,!\,E$ and $M \Downarrow^{V}_{\Delta,\iota} (V, \Delta')$ for some $\mathrm{wf}(\Delta)$, then $(\!|M|\!) \Downarrow_{\downarrow\Delta,\iota} ((\!|V|\!), \downarrow\Delta')$

5 Sequenced Joins

Queries that join multiple tables are straightforward to encode using language integrated query. Keeping with our employee database, say we wish to separate out the salary into a separate table. The non-temporal employee database might look as follows:

employees

name	position	band
Alice	Senior Lecturer	A08
Bob	PhD Student	B01
Charles	PhD Student	B01
Dolores	Professor	A10

salaries

band	salary
A08	40000
A09	50000
A10	70000
B01	15000

We can get the salary for each employee as follows:

```
query
  for (e ← get employees)
    for (s ← get salaries)
      where (e.band = s.band)
      ≀(name = e.name, salary = s.salary)∫
```

name	salary
Alice	40000
Bob	15000
Charles	15000
Dolores	70000

Joining a temporal table with a non-temporal table is also easily expressible. Consider a version of our previous temporal employees table from just after when Dolores joined:

name	position	band	start	end
Alice	Lecturer	A08	2010	2018
Alice	Senior Lecturer	A09	2018	∞
Bob	PhD Student	B01	2019	2023
Charles	PhD Student	B01	2018	2022
Dolores	Professor	A10	2022	∞

We can join this table with the non-temporal salaries table as follows; for clarity, we denote valid-time **get** as $\mathbf{get}_{\mathsf{V}}$:

```
query
  for (e ← get_V employees)
    for (s ← get salaries)
      where ((data e).band = s.band)
      ≀(name = e.name, salary = s.salary)[start e,end e)∫
```

giving us the corresponding table in Section 4.

Things get more interesting when *both* tables are temporal. Salaries are not static over time; bands go up with inflation, for example. Suppose we now have two temporal tables. Consider the above table along with a temporal salaries table showing a pay increase in 2015:

band	salary	start	end
A08	38000	2000	2015
A09	48000	2000	2015
A08	40000	2015	∞
A09	50000	2015	∞
···	···	···	···

What does it mean to join two *temporal* tables? In essence, we want to record *all* configurations of a particular joined record, creating new records with shorter periods of validity whenever data from either underlying table changes. Concretely, joining the above two temporal tables would give:

name	salary	start	end
Alice	38000	2010	2015
Alice	40000	2015	2018
Alice	50000	2018	∞
···	···	···	···

Now there are records for Alice for *three* different periods:

- The first when Alice was on salary band A08, conferring a salary of £38000.
- The second when band A08 increased to £40000.
- The third when Alice was promoted to band A09.

Such joins are called *sequenced* because they (conceptually) evaluate the join on the whole sequence of states encoded by each table. Manually writing the sequenced joins in SQL is error-prone. We instead introduce a construct, **join**, which allows us to write the following:

```
join
  for (e ← getᵥ employees)
    for (s ← getᵥ salaries)
      where ((data e).band = (data s).band)
      ⟅(name = (data e).name, salary = (data s).salary)⟆
```

Note that we *do not* need to calculate the period of validity for each resulting row; this is computed automatically.

Figure 9 shows how sequenced joins can be implemented; we show the constructs for valid time, but the same technique can be used for transaction time. The typing rule requires that the result of a **join** query is *flat* (nested sequenced queries are conceptually nontrivial).

As mentioned earlier, queries can be rewritten to normal forms for conversion to SQL, as shown in Figure 9. The structure of these normal forms allows sequenced joins to be implemented through a simple rewrite: the **greatest** and **least** functions are used to calculate the intersections of the periods of validity for each combination of records from each generator, with the modified predicate ensuring that the periods of overlap make sense. The calculated overlapping periods of validity are then returned in the resulting row.

6 Implementation and Case Study

The Links programming language [9] is a statically-typed functional web programming language which allows client, server, and database code to be written in a uniform language. We have extended Links with support for the constructs described in Sections 3 and 4, as well as support for temporal

Typing rules $\boxed{A :: \text{FQType}}$ $\boxed{\Gamma \vdash M{:}A\,!\,E}$

$$\frac{}{C :: \text{FQType}} \qquad \frac{}{(\ell_i : C_i)_i :: \text{FQType}}$$

$$\frac{\Gamma \vdash M{:}\text{Bag}(A)\,!\,E \qquad A :: \text{FQType} \qquad E \subseteq \{\textbf{read}\}}{\Gamma \vdash \textbf{join}\ M{:}\text{Bag}(\text{ValidTime}(A))\,!\,E}$$

Normal forms

Queries	Q	::=	$K_1 \uplus \cdots \uplus K_n$
Comprehensions	K	::=	$\textbf{for}\ (\widetilde{G})\ \text{where}\ P\ \lceil S \rfloor$
Generators	G	::=	$x \leftarrow \textbf{get}\ t \mid x \leftarrow \textbf{get}_{\text{v}}\ t$
Normalised terms	S	::=	$P \mid R$
Base terms	P	::=	$c \mid x.\ell \mid \odot\{\overrightarrow{P}\}$
Record terms	R	::=	$(\ell_i = P_i)_i$

Translation on normal forms

$$\|\textbf{join}\ Q\| = \textbf{query}\ \|Q\| \qquad \|K_1 \uplus \cdots \uplus K_n\| = \|K_1\| \uplus \cdots \uplus \|K_n\|$$

$$\|P\| = P \qquad \|R\| = R$$

$$\left\| \begin{array}{l} \textbf{for}\left(\begin{array}{l} x_1 \leftarrow \textbf{get}_{\text{v}}\ t_1, \ldots, x_m \leftarrow \textbf{get}_{\text{v}}\ t_m, \\ y_1 \leftarrow \textbf{get}\ \ t'_1, \ldots, y_n \leftarrow \textbf{get}\ \ t'_n \end{array}\right) \\ \quad \text{where}\ (P) \\ \quad \lceil R \rfloor \end{array} \right\| =$$

$$\textbf{for}\left(\begin{array}{l} x_1 \leftarrow \textbf{get}_{\text{v}}\ t_1, \ldots, x_m \leftarrow \textbf{get}_{\text{v}}\ t_m, \\ y_1 \leftarrow \textbf{get}\ \ t'_1, \ldots, y_n \leftarrow \textbf{get}\ \ t'_n \end{array}\right)$$
$$\text{where}\ (P \wedge \mathit{joinStart} < \mathit{joinEnd})$$
$$\lceil R^{[\mathit{joinStart}, \mathit{joinEnd})} \rfloor$$

$$\text{where}\quad \mathit{joinStart} \triangleq \textbf{greatest}(x_1.\mathit{start}, \ldots, x_m.\mathit{start})$$
$$\mathit{joinEnd} \triangleq \textbf{least}(x_1.\mathit{end}, \ldots, x_m.\mathit{end})$$

Figure 9. Sequenced joins

joins as described in Section 5. In this section, we describe a case study based on curating COVID-19 data.

Our translations from λ_{VLINQ} into λ_{LINQ} are trivially realisable in SQL. Queries can be compiled using known techniques (e.g., [8]). The startRows and endRows functions can be compiled using an SQL `WITH` statement, and there is a direct correspondence between λ_{LINQ} modification operations and their SQL equivalents. Each translated temporal modification is executed as an SQL transaction, with primary key and referential integrity constraint checking deferred until the end of the transaction.

Case study. We have used the temporal features of Links in two prototypes based on curated scientific databases: curation of publicly available COVID-19 data, and storage and curation of XML documents [18] using the Dynamic Dewey labelling algorithm [38].

We concentrate on the first application[1]; a previous prototype which used a preliminary version of the language design has previously been presented as a short demo paper [17]. In 2020, the Scottish Government began releasing various data about the COVID-19 pandemic [26], which included weekly data of fatalities in Scotland in various categories (such as

[1]Links code is available at https://github.com/vcgalpin/links-covid-curation [16]

'Sex'). Each weekly release was a CSV file, with a row for each subcategory (e.g., 'Sex' has the subcategories 'Male' and 'Female') and a column for each week for which data was available. Each release included an additional week column with the latest data (see Figure 10). Importantly, each release could include revisions to data for previous weeks.

Information about the changes to the data over time is often desired to understand its provenance and assess its trustworthiness [4]. From a provenance point of view, this data is interesting because a column for an earlier week may contain updated data. We developed a web application for the querying of the data ("How do the Male and Female subcategories compare in terms of the change in fatalities from last week to this week?") as well as querying the changes in the data ("How do the Male and Female subcategories compare in terms of number of updates to existing values?").

Considering the non-temporal data, an entry in a database table would be a row consisting of the key fields `subcat` and `weekdate` and a value field giving the corresponding `count`. In the case of the temporal data, the key fields are insufficient to uniquely identify the value of the `count` because it may have different values over time. Thus the time validity fields are necessary to provide a key for the value.

The prototype uses a valid time table for fatality data to capture the notion that a count value, either brand new or an update, becomes valid as soon as the CSV is uploaded into the interface[2] (this can be a different time from when the new value is accepted and written to the database). In Links it is possible to specify the names of the period stamping fields, which have the built-in type `DateTime`. This table is defined using the following Links code; we have omitted some details in the code snippets for brevity.

```
var covid_data =
  table "covid_data"
    with (subcat: Int, weekdate: String, count: Int)
    using valid_time(valid_from, valid_to)
    from database "covid_curation";
```

The prototype's upload workflow is as follows: the user uploads a new CSV file, and the count values for the new week are added to the database. For counts that pertain to earlier weeks and that now have different values, the user is shown these counts and can accept them, reject them or move them to a pending list for a later decision. In terms of implementation, brand new count values are added to the table with a *sequenced insert*, using the upload time as the start time. The Links code for this and other examples can be found in the extended version. The process is more complex for updated count values, because the interface shows the user previous value, to support decision making. This requires a conditional join over the current state of the `covid_data` table and the count values from the CSV file. If a modification is accepted, it is added using a *sequenced update*. Figure 10 illustrates how the table changes as a result of a single update.

The prototype also provides functionality to query data, both as current data, and as data with information about changes. The current data is obtained using a *current query*. The result is a list of weeks and counts grouped by subcategory. This is repeated for each category.

```
fun getCurrentData (category) {
  query nested {
    for (x <-- subcategory)
      where (x.cat == category)
      [(subcat_name = x.subcat_name, cat = x.cat,
        results =
          for (y <- vtCurrent(covid_data))
            for (z <-- week)
              where (y.subcat == x.subcat &&
                     y.weekdate == z.weekdate &&
                     z.all_zero == false)
              [(count = y.count, weekdate = y.weekdate)])]}
}
```

Instead of an explicit **get** construct, Links uses the 'double arrow' comprehension `<--` to represent a nontemporal database query, with `<-t-` and `<-v-` supporting transaction time and valid time queries respectively. The 'single arrow' comprehension `<-` denotes a list comprehension. Finally, `vtCurrent` is a standard library function which performs a valid time query to obtain the values valid at the current time.

For update provenance queries of individual counts, a self join is computed over the `subcategory` and `week` fields of the valid time table to provide a nested result table where each count is associated with a list of count values and their associated start and end time information. This is a nonsequenced query because the time period information is explicitly added to the result table. The user can specify the subcategory and week they are interested in, and obtain details of modifications. The interface also supports update provenance by week and by category. This is illustrated in Figure 11.

7 Discussion

Efficiency. Our main focus has been on *portability*: by distilling a language design and formalising and implementing a translation from temporal calculi to non-temporal calculi, we allow temporal functionality to be used on a mainstream DBMS. The cost of portability is that our translations will inevitably not perform as well as a native implementation. Although we do not make any specific claims about efficiency, we have no reason to believe that the performance is any different to hand-translated SQL.

In particular, as discussed in §6, all translated queries can be run directly on the database and do not require in-memory processing. Previous work on language-integrated query (e.g., [6, 9]) shows how the "nested loop" style of query is translated into efficient SQL, and our translation of queries

[2]Other possibilities for the start time of validity are the date of the release of the CSV file or the start of the new week.

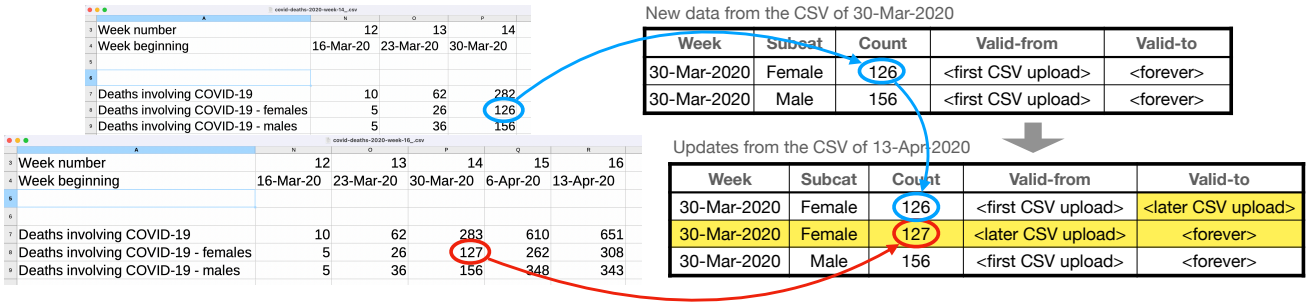


Figure 10. Example of data uploads, sequenced insertion and sequenced update

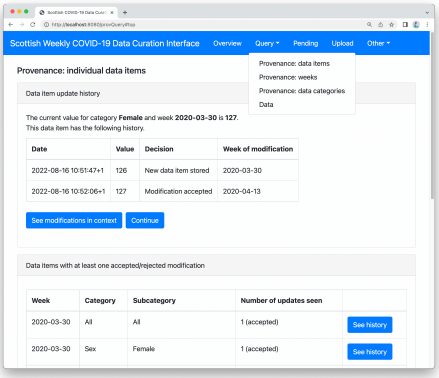


Figure 11. Interface screenshot: history of a count

happens prior to normalisation. Further optimisation is subsequently performed by the DBMS, and anecdotally we have not observed any performance issues in any applications we have written using the temporal extensions to Links.

Supporting existing native implementations. Some (mainly proprietary) DBMSs, for example Teradata [1], have native support for some temporal features inspired by TSQL2 or SQL:2011. Although we are yet to explore this, an advantage of the LINQ approach is that temporal SQL syntax could be generated for backends which support temporal operations directly, while maintaining functionality in mainstream backends without native temporal support.

The translation between our temporal modification constructs and SQL:2011 (and similarly, TSQL2) would be fairly direct. Consider a sequenced update in λ_{VLINQ}:

update sequenced $(x \Leftarrow employees)$
between $2022{-}01{-}1$ **and** $2022{-}10{-}29$ **where** $(x.salary < 30000)$
set $(salary = x.salary + 1000)$

This could be implemented in SQL:2011 as follows:

```
UPDATE employees
  FOR PORTION OF EPeriod
    FROM DATE '2022-01-01' TO DATE '2022-10-29'
  WHERE salary < 30000
  SET salary = salary + 1000
```

Moving between the two DBMSs would not require any changes to application source code. However, not all operations can be as straightforwardly translated: in particular, SQL:2011 does not natively support sequenced joins.

8 Related and Future Work

Most of the focus of effort on language-integrated query has been (perhaps unsurprisingly) on queries rather than updates, beginning with the foundational work on nested relational calculus by Buneman et al. [3] and on rewriting queries for translation to SQL by Wong [37]. Lindley and Cheney [23] presented a calculus including both query and update capabilities and our type and effect system for tracking database read and write access is loosely based on theirs. More recently a number of contributions extending the formal foundations of language-integrated query have appeared, including to handle higher-order functions [8], nested query results [6], sorting/ordering [21], grouping and aggregation [27, 28], and deduplication [30]. Our core calculus λ_{LINQ} only incorporates the first two of these, and developing a core calculus that handles more features, as well as translating temporal queries involving them, is an obvious future direction. To the best of our knowledge no previous work on language-integrated query has considered temporal data specifically.

The ubiquity and importance of time in applications of databases was appreciated from an early stage [7] and led to a significant community effort to standardise temporal extensions to SQL based on the TSQL2 language design in the 1990s and early 2000s [33]. This effort ultimately resulted in standardisation of a relatively limited subset of the original proposal in SQL:2011 [22]. Since then temporal database research has progressed steadily, including recent contributions showing how to implement temporal data management as a layer on top of a standard RDBMS [11], and establishing connections between temporal querying and data provenance and annotation models [12].

Snodgrass [34] describes how to implement TSQL2-style updates and queries by translation to SQL, but we are not aware of previous detailed formal proofs of correctness of

translations for transaction time and valid time updates. Although timestamping rows with time intervals is among the most popular ways to represent temporal databases as flat relational tables, it is not the only possibility. Jensen et al. [20] proposed a *bitemporal conceptual data model* that captures the abstract meaning of a temporal table and used it to compare different representation strategies.

There can be multiple representations of the same abstract temporal data, leading to consideration of the problem of coalescing or normalizing the intervals to save space and avoid ambiguity. Nonsequenced updates can be used to perform modifications that have different effects on representations of the same conceptual table. We have not considered coalescing or other common issues such as how to handle operations such as deduplication, grouping and aggregation (including emptiness testing), or integrity constraints in a temporal setting. Some of these issues appear orthogonal to the high-level language design and could be incorporated "under the hood" into the implementation or even performed directly on the database.

One important future application is to retrofit temporal aspects to expert-curated databases, an example being the Guide to Pharmacology Database (GtoPdb) that summarises pharmacological targets and interactions [2]. Links has been used to implement a workalike version of GtoPdb [15] and we hope to build on this to provide a fully versioned implementation of GtoPdb. An important requirement here is to minimise changes to the existing system.

Finally we mention two immediate next steps. First, we plan to investigate *bitemporal* databases [35], which allow transaction and valid time to be used together, in turn allowing us to write queries such as "when was it recorded that Bob's contract length was extended?". Bitemporal databases are considerably more difficult to formalise and reason about, so we aim to investigate how bitemporal support can be added in a compositional manner. Second, at present, the result of a sequenced join must be a flat record; further work is required to understand the semantics and implementation techniques for joins that produce nested results.

9 Conclusions

In spite of decades of work on temporal databases and even an extension to the SQL standard, mainstream support for temporal data remains limited, requiring developers to implement temporal functionality from scratch. In this paper, we have shown how to extend language-integrated query to support transaction time and valid time data, making temporal data management accessible without explicit DBMS support. We have formalised our constructs and translational implementation strategies based on those proposed by Snodgrass [34], and proved that the translations are semantics-preserving. We have implemented our approach in the Links programming language and assessed its value through a case study. Our work is a first but significant step towards fully supporting temporal data management at the language level.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. This work is partially funded by EPSRC Grant EP/T014628/1 (STARDUST), ERC Consolidator Grant 682315 (Skye), and a UK Government ISCF Metrology Fellowship.

References

[1] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. 2013. Temporal query processing in Teradata. In *EDBT*. ACM, 573–578.

[2] J.F. Armstrong, E. Faccenda, S.D. Harding, A.J. Pawson, C. Southan, J.L. Sharman, B. Campo, D.R. Cavanagh, S.P.H. Alexander, A.P. Davenport, M. Spedding, and J.A. Davies. 2020. The IUPHAR/BPS Guide to PHARMACOLOGY in 2020: extending immunopharmacology content and introducing the IUPHAR/MMV Guide to MALARIA PHARMACOLOGY. *Nucleic Acids Research* 48 (2020), D1006–D1021. https://doi.org/10.1093/nar/gkz951

[3] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.* 149, 1 (1995). https://doi.org/10.1016/0304-3975(95)00024-Q

[4] Peter Buneman and Wang-Chiew Tan. 2018. Data Provenance: What next? *SIGMOD Rec.* 47, 3 (2018), 5–16. https://doi.org/10.1145/3316416.3316418

[5] James Cheney, Sam Lindley, and Philip Wadler. 2013. A practical theory of language-integrated query. In *ICFP*. https://doi.org/10.1145/2500365.2500586

[6] James Cheney, Sam Lindley, and Philip Wadler. 2014. Query shredding: efficient relational evaluation of queries over nested multisets. In *SIGMOD*. ACM, 1027–1038. https://doi.org/10.1145/2588555.2612186

[7] James Clifford and David S. Warren. 1983. Formal Semantics for Time in Databases. *ACM Trans. Database Syst.* 8, 2 (1983), 214–254. https://doi.org/10.1145/319983.319986

[8] Ezra Cooper. 2009. The Script-Writer's Dream: How to Write Great SQL in Your Own Language, and Be Sure It Will Succeed. In *DBPL (Lecture Notes in Computer Science, Vol. 5708)*. Springer, 36–51. https://doi.org/10.1007/978-3-642-03793-1_3

[9] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *FMCO (Lecture Notes in Computer Science, Vol. 4709)*. Springer, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

[10] George Copeland and David Maier. 1984. Making Smalltalk a database system. *SIGMOD Rec.* 14, 2 (1984). https://doi.org/10.1145/602259.602300

[11] Anton Dignös, Michael H. Böhlen, Johann Gamper, and Christian S. Jensen. 2016. Extending the Kernel of a Relational DBMS with Comprehensive Support for Sequenced Temporal Queries. *ACM Trans. Database Syst.* 41, 4, Article 26 (2016), 46 pages. https://doi.org/10.1145/2967608

[12] Anton Dignös, Boris Glavic, Xing Niu, Michael Böhlen, and Johann Gamper. 2019. Snapshot Semantics for Temporal Multiset Relations. *Proc. VLDB Endow.* 12, 6 (2019), 639–652. https://doi.org/10.14778/3311880.3311882

[13] Stefan Fehrenbach and James Cheney. 2018. Language-integrated provenance. *Sci. Comput. Program.* 155 (2018), 103–145. https://doi.org/10.1016/j.scico.2017.08.009

[14] Simon Fowler, Vashti Galpin, and James Cheney. 2022. Language-Integrated Query for Temporal Data (Extended version). https://doi.org/10.48550/ARXIV.2210.12077

[15] Simon Fowler, Simon Harding, Joanna Sharman, and James Cheney. 2020. Cross-tier Web Programming for Curated Databases: A Case Study. *International Journal of Digital Curation* 16, 1 (2020). https://doi.org/10.2218/ijdc.v15i1.717

[16] Vashti Galpin. 2022. vcgalpin/links-covid-curation: Initial release (v0.1.0). Zenodo. https://doi.org/10.5281/zenodo.7199221.

[17] Vashti Galpin and James Cheney. 2021. Curating Covid-19 Data in Links. In *IPAW 2020 + IPAW 2021 (Lecture Notes in Computer Science, Vol. 12839)*. Springer, 237–243. https://doi.org/10.1007/978-3-030-80960-7_19

[18] Vashti Galpin, Ian Smith, and Jean-Laurent Hippolyte. 2022. Supporting provenance of digital calibration certificates with temporal databases. In *IMEKO TC6 M4Dconf.* To appear.

[19] Christian S. Jensen and Richard T. Snodgrass. 1999. Temporal Data Management. *IEEE Trans. Knowl. Data Eng.* 11, 1 (1999), 36–44. https://doi.org/10.1109/69.755613

[20] Christian S. Jensen, Michael D. Soo, and Richard T. Snodgrass. 1994. Unifying Temporal Data Models via a Conceptual Model. *Inf. Syst.* 19, 7 (1994), 513–547. https://doi.org/10.1016/0306-4379(94)90013-2

[21] Oleg Kiselyov and Tatsuya Katsushima. 2017. Sound and Efficient Language-Integrated Query - Maintaining the ORDER. In *APLAS 2017*. 364–383. https://doi.org/10.1007/978-3-319-71237-6_18

[22] Krishna Kulkarni and Jan-Eike Michels. 2012. Temporal Features in SQL:2011. *SIGMOD Rec.* 41, 3 (2012), 34–43. https://doi.org/10.1145/2380776.2380786

[23] Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *TLDI*. https://doi.org/10.1145/2103786.2103798

[24] John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *POPL*. ACM Press, 47–57. https://doi.org/10.1145/73560.73564

[25] Erik Meijer, Brian Beckman, and Gavin M. Bierman. 2006. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD*. https://doi.org/10.1145/1142473.1142552

[26] National Records of Scotland. 2021. Deaths involving coronavirus (COVID-19) in Scotland: archive. https://www.nrscotland.gov.uk/statistics-and-data/statistics/statistics-by-theme/vital-events/general-publications/weekly-and-monthly-data-on-births-and-deaths/deaths-involving-coronavirus-covid-19-in-scotland/archive Last accessed: 23 April 2021.

[27] Rui Okura and Yukiyoshi Kameyama. 2020. Language-Integrated Query with Nested Data Structures and Grouping. In *FLOPS*. 139–158. https://doi.org/10.1007/978-3-030-59025-3_9

[28] Rui Okura and Yukiyoshi Kameyama. 2020. Reorganizing Queries with Grouping. In *GPCE* (Virtual, USA). ACM, 13 pages. https://doi.org/10.1145/3425898.3426960

[29] Quill team. 2022. Quill: Compile-Time Language Integrated Queries for Scala. Open source project. https://github.com/getquill/quill.

[30] Wilmer Ricciotti and James Cheney. 2021. Query Lifting: Language-integrated query for heterogeneous nested collections. In *ESOP*. 579–606. https://doi.org/10.1007/978-3-030-72019-3_21

[31] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. 1988. Extended Algebra and Calculus for Nested Relational Databases. *ACM Trans. Database Syst.* 13, 4 (1988), 389–417. https://doi.org/10.1145/49346.49347

[32] Lwin Khin Shar and Hee Beng Kuan Tan. 2013. Defeating SQL Injection. *Computer* 46, 3 (2013), 69–77. https://doi.org/10.1109/MC.2012.283

[33] Richard T. Snodgrass (Ed.). 1995. *The TSQL2 Temporal Query Language*. Kluwer.

[34] Richard T. Snodgrass. 1999. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann.

[35] Richard T. Snodgrass and Ilsoo Ahn. 1985. A Taxonomy of Time in Databases. In *SIGMOD Conference*. ACM Press, 236–246.

[36] Don Syme. 2006. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *ML Workshop*.

[37] Limsoon Wong. 1996. Normal Forms and Conservative Extension Properties for Query Languages over Collection Types. *J. Comput. Syst. Sci.* 52, 3 (1996). https://doi.org/10.1006/jcss.1996.0037

[38] Liang Xu, Tok Wang Ling, Huayu Wu, and Zhifeng Bao. 2009. DDE: from dewey to a fully dynamic XML labeling scheme. In *SIGMOD*. ACM, 719–730. https://doi.org/10.1145/1559845.1559921

Received 2022-08-12; accepted 2022-10-10