An Approach for Persistent Time-Varying Values

Tetsuo Kamina
Oita University
Japan
kamina@acm.org

Tomoyuki Aotani
Tokyo Institute of Technology
Japan
aotani@c.titech.ac.jp

Abstract

As reactive systems, such as cyber-physical systems and the Internet of Things, become increasingly important, time-varying values, also known as *signals*, are playing an important role in software development. Although reactive systems require the change histories of some signals to be stored for various purposes such as post analysis and simulation, current programming languages do not provide a way to declare that signals are persistent. This paper proposes a method that realizes persistent signals in a reactive programming language, where (1) every update to each persistent signal is recorded in a time-series database, which can be seen as a part of the programming language runtime; and (2) persistent signals support a convenient time-oriented query mechanism. In this approach, each signal in the reactive programming language is seamlessly connected with the time-series database. This method is implemented as an extension of SignalJ, a Java-based reactive programming language that supports signals. In the implementation, the persistent signal mechanism is integrated with TimescaleDB, a PostgreSQL-based time-series database. In preliminary performance evaluations, our implementation had good responsiveness on most tests, indicating its feasibility for use in many applications.

CCS Concepts **• Software and its engineering → Language features**; **• Information systems →** *Database design and models.*

Keywords Signals; Reactive programming; Time-series databases

ACM Reference Format:
Tetsuo Kamina and Tomoyuki Aotani. 2019. An Approach for Persistent Time-Varying Values. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '19), October 23–24, 2019, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3359591.3359730

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Onward! '19, October 23–24, 2019, Athens, Greece
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6995-4/19/10...$15.00
https://doi.org/10.1145/3359591.3359730

1 Introduction

Reactive systems are becoming increasingly important, and time-varying values, also known as *signals*, play an important role in such systems. Each signal can be viewed as a data stream with a periodically updated value. By "connecting" signals in a functional manner, we can declaratively specify a dataflow with inputs given by the environment and outputs that respond to the changes in the environment. Modern applications such as cyber-physical systems (CPS) and the Internet of Things (IoT) are naturally represented by the use of signals. There have been many reactive programming (RP) languages that directly support signals [4, 7, 14, 18, 24, 29].

Reactive systems require that some signals be persistent. For example, in a vehicle tracking system, the position of each vehicle, the status of each traffic light, and the condition of each street are characterized by time-varying values, and these are also considered persistent in that the values should be recorded on the disk to allow rewinding to any earlier point in the execution history for inspection (e.g., inspecting the cause of a car accident). By contrast, some time-varying values can be computed from persistent signals (e.g., the existence of a traffic jams) and so it is not necessary for them to be persistent.

Unfortunately, signals in existing RP languages are not persistent. Although time-traveling debuggers [21, 25] record the execution histories of all signals in the application to enable programmers to pause execution and rewind to any earlier execution point, such tools are designed for debugging; thus, they are not suitable for use as the application framework that supports persistent signals. In those tools, the change histories of all signals are implicitly stored on the disk only when the application is running in the debugging mode. To our knowledge, useful programming interfaces for software development (not for debugging) that allow issuing queries over signals whose change histories are stored on a disk have not been considered to date.

From the viewpoint of database systems, persistent time-varying values are closely related to time-series databases [13] in that such databases store time-series data (i.e., change histories of time-varying values). Such databases are specialized to store time-series data that are indexed by their timestamps, and the database system supports a convenient time-oriented

API and a compact representation of the data and time indices. However, there is a significant gap between signals and time-series data. Database languages adopt a computation model that is quite different from the model used to compute the values of the signals. For example, recomputation and update propagation are rarely considered in database languages. Furthermore, there is a significant mismatch between how to handle *time*; while signals handle logical time, time-series data are typically related to real time.

Our goal in the study described here is to provide a programming interface in which a persistent time-series data mechanism is fully integrated with signals. We call such signals *persistent signals*. Persistent signals have the same capabilities as non-persistent signals; i.e., they can be connected with other signals to represent a dataflow and can be updated by means of update propagations. Internally, every update to a persistent signal is recorded in the database, which can be seen as a part of the proposed programming language runtime. Each persistent signal supports a query mechanism, which is represented using the RP language API, to inspect the value of the signal at any earlier execution point. The programmers need not explicitly consider the database. The database schema is automatically generated from the source code and modified, if necessary, when the source code is modified.

To study the feasibility of this goal, we extended SignalJ [14], a Java-based RP language that supports signals, to realize persistent signals. Its language runtime is also extended to provide a wrapper for the time-series database so as to realize a natural API for queries over persistent signals. This wrapper is implemented using TimescaleDB[1], an open-source time-series database based on PostgreSQL. The wrapper manages the conversions from the API to queries in the database language, including the mapping from logical time to real time. In this setting, we conducted a series of preliminary performance evaluations, which shows that our implementation is responsive on most tasks, indicating its feasibility for use in many applications.

The rest of this paper is structured as follows. Section 2 introduces SignalJ. Section 3 gives the motivation for this proposal, using the example of a vehicle-tracking system. Section 4 gives the design for extending SignalJ to handle persistent signals and the API of the extension. Section 5 explains the details of how this proposal is implemented. Section 6 shows the results of preliminary evaluations. Section 7 discusses related work, and Section 8 concludes this paper and presents some directions for future work.

2 Signals in SignalJ

RP is a programming approach in which change propagations through time-varying values, each of which constitutes a data stream with its own periodically updated value, are declaratively specified. A time-varying value, also known as a *signal*, can be handled using a function. For example, assuming that the power difference of an actuator is calculated by the function `f` that takes a sensor value as input, both the power difference and the sensor value can be represented as signals, `powerDifference` and `sensorValue`, respectively, and the relationship between them can be simply represented by the following declaration.

```
powerDifference = f(sensorValue)
```

This declaration specifies that every update of `sensorValue` results in a recalculation of `powerDifference`. This mechanism is present in by several functional-reactive programming languages and directives such as Fran [7], FrTime [4], and Flapjax [18].

Recently, this mechanism has been included in imperative object-oriented languages such as REScala [24] and SignalJ [14] where imperative updating of signals is allowed. In SignalJ, a signal is declared using the modifier `signal`. For example, the following code fragment declares two signals, `a` and `b`, where `b` depends on `a`.

```
signal int a = 5;
signal int b = a + 3;
a++;
System.out.println(b); // displays 9
```

We refer to a signal that depends on other signals as a *composite signal* (a signal depends on all signals that appear on the right-hand side of the initialization (=) of the signal, and this dependency is transitive). This means that, when the value of `a` is updated, this update is propagated to `b`, resulting in the value of `b` being updated. Thus, the value of `b` in the above code fragment is initially 8, but if the value of `a` is updated by `a++`, the value of `b` becomes 9.

This dependency between `a` and `b` is fixed for the duration of execution. This means that any reassignment of a value to `b` is not allowed in SignalJ: the value of `b` can be updated only through updating `a`. On the other hand, `a` does not depend on any other signal. We refer to this signal as a *source signal*, which can be *imperatively* updated (as performed by `a++`).

A notable feature of SignalJ is how it distinguishes signals from non-signals. In SignalJ, variables that have the modifier `signal` are signals; those that do not use it are not signals. The types `signal int` and `int` are the same base type, which means a signal can be used anywhere that a non-signal is expected. This approach is especially useful for creating a dependency network of signals using legacy Java libraries. For example, the following code fragment counts the number of 1-bits in the 2's complement representation of the signal `a`:

```
signal int a = 0;
signal int count = Integer.bitCount(a);
```

[1] https://www.timescale.com

The method `bitCount` is provided by the Java standard library. It takes an `int` value as an argument and returns an `int` value. Because `a` can be used wherever an `int` value is expected, it can be provided as the argument to `bitCount`. Because `count` is declared as a signal, when `a` is updated, `bitCount` is automatically recomputed to update `count`. In other words, the signal dependency is determined from the lexical scope of the signal declaration.

SignalJ also supports an event mechanism, where an event is an update of a signal; that is, we can implement an event handler that responds to an update of a signal. This event handler is a lambda expression (or method reference) that is passed to the `subscribe` operator of the signal. The following code fragment shows an example.

```
signal int a = 5;
a.subscribe(e -> System.out.println(e));
a++; // displays 6
```

The handler is called whenever the signal is updated. Thus, the lambda expression passed to `subscribe` is called at the subsequent `a++`, and the value of `a`, which is now 6, is displayed. This event mechanism is useful for representing the side-effects of signal updates (e.g., actuating a motor), which is common when implementing IoT applications using legacy libraries.

3 Making Signals Persistent

Signals directly represent dataflows from inputs given by the environment to outputs that respond to the changes in the environment. This feature is useful in implementing modern reactive software such as CPS and IoT applications.

However, one important building block is still missing in existing RP languages: storing the values of signals in the database according to time. We explain this using the example of a vehicle tracking system. This system records the position of each vehicle, which is obtained from automotive devices. The position changes while the vehicle is moving; thus, it is a time-varying value. There are also some other time-varying values that depend on the position, such as the estimated velocity and the total traveled distance of that vehicle. These dependencies on time-varying values motivate us to develop a system to handle them using an RP language.

This vehicle tracking system also allows for post analysis (e.g., inspecting the cause of a car accident) and simulation. To accomplish this, the change history of each position of the vehicle is stored in the database. There are several time-series databases that are useful for this purpose, such as OpenTSDB[2], InfluxDB[3], and TimescaleDB. These databases are specialized to store time-series data. Such database systems typically provide a compact representation and convenient time-oriented API for time-series data.

[2]http://opentsdb.net
[3]https://www.influxdata.com

However, these database systems are not easily interoperable with RP languages, and it is still unclear how to transparently use such time-series data from signal networks in an RP language. The database languages assume a computation model that is quite different from how the signals are computed. For example, recomputation and update propagation are rarely considered in database languages. Programmers must manually bridge this gap (e.g., by means such as event mechanisms).

There is also a significant mismatch between how time is handled in the databases and in the RP language. Basically, time-series databases handle the *real time*. For example, values may be monitored by several distinct sensors operating in parallel, and the values are stored in the database with their timestamps. Because of delays caused by serializing these disk accesses, the timestamps of two simultaneous writing can be distinct. However, we consider such values to have been observed at the same time, and the signals in the RP language should reflect this fact. In other words, signals handle *logical time*

This paper proposes a mechanism of *persistent signals*, where the time-series data is transparently perceived through the signals. To achieve transparency between the programming language interface and the underlying database, the following features are provided.

- A persistent signal is declared as a variant of signals that encapsulates details of the underlying database. All queries of the database are performed through the API, and programmers do not have to know what internal queries are issued to the underlying database.
- Internally, each update to the persistent signal is assigned a serial number that corresponds to the logical time, along with the provided timestamp. The timestamp is used to execute time-oriented queries, and the serial number is used to identify which updates occured at the same time.

Additionally, we include the following features to ensure that the proposed mechanism fits well with the signal mechanism of SignalJ.

- Each persistent signal is compatible with a non-persistent signal; that is, a persistent signal can appear in any place where a non-persistent signal is expected, and any operations available for a non-persistent signal are also available for a persistent one. In addition, some specific operations are provided for persistent signals (such as queries over the update histories).
- Composite signals that depend on persistent source signals are not necessarily stored on the disk. This is because the composite signals are recomputed using the source signals. For example, the update history of the velocity of the vehicle is not necessarily stored as time-series data because this can be calculated from the update history of the position of the vehicle.

4 Language Design

To realize persistent signals, we extend SignalJ. In addition to signals, this extension provides two language features: (1) a signal can be declared as a persistent signal explicitly; and (2) several time-oriented operations are provided for persistent signals. Syntactically, this extension adds only two modifiers, `persistent` and `nonpersistent`, which indicate the signal is persistent or non-persistent, respectively. Other language features are achieved by providing an API to handle persistent signals and their time-oriented operations. This API is summarized in Table 1.

4.1 Persistent Signals

A *persistent signal* is a signal whose update history is recorded in the database. A signal is declared as a persistent signal using the modifier `persistent`. In the following example, `car1234_x` and `car1234_y` are declared as persistent signals whose time-varying values are of type `int`.

```
persistent signal int car1234_x, car1234_y;
signal int dx = car1234_x.lastDiff(1);
signal int dy = car1234_y.lastDiff(1);
signal int v = dx.distance(dy);
```

The declaration of a persistent signal binds the declared signal with a corresponding database component, such as a table in an RDB. In the above example, these persistent signals represent the position of a specific vehicle: `car1234_x` represents the x-coordinate and `car1234_y` represents the y-coordinate.

While the above vehicle is moving, the values of the persistent signals are updated using the values obtained from automotive devices. The signal `v` in the above code fragment represents the estimated velocity of the vehicle, which is recalculated each time the position of the vehicle changes. The method `lastDiff` returns the difference between the current value and the last value of the receiver signal (if its execution history is empty or only one record has been stored in the history, this method returns a default value; in this case, 0 is used). When both `car1234_x` and `car1234_y` are updated, the results of `lastDiff`, that are, `dx` and `dy`, respectively, are also updated. Thus, both `dx` and `dy` are also declared as signals. Assuming that these signals are updated periodically, the velocity `v` of the vehicle can be determined by calculating the distance expressed by `dx` and `dy`. Because `v` is also a time-varying value, it is also declared as a signal that depends on `dx` and `dy`.

We note that the signals `dx`, `dy`, and `v` depend on the persistent signals. This means that even though they are not declared with `persistent`, their execution histories can be derived from the source persistent signals. We call such signals as *view signals*. In our language, some methods of persistent signals exist for view signals also (Table 1). Our compiler implicitly infers whether the declared signal is a view signal by traversing the right-hand side of the signal declarations. To disable this inference for a specific signal, we need to declare that signal using the modifier `nonpersistent`. Currently, only signals that depend on persistent signals and signals that do not depend on any other signals (except `nonpersistent` signals) can be declared as `nonpersistent`.

One issue that we need to consider is that these coordinates represent positions that are monitored *simultaneously*. Because the updates of signals may occur independently, these updates can occur at different real times, and thus it is necessary to let the language runtime know that the update time of these signals should be considered the same. Our language considers the updates of persistent signals declared in parallel (e.g., `car1234_x` and `car1234_y` in the above example) to occur simultaneously. For example, the following loop updates both `car1234_x` and `car1234_y` periodically, and our language treats the two assignments below as occurring simultaneously.

```
while (true) {
  car1234_x = getPosition().x;
  car1234_y = getPosition().y;
  Thread.sleep(5000);
}
```

It is the programmer's responsibility to ensure that both `car1234_x` and `car1234_y` are updated only in the same loop. This means that simultaneous updates of `car1234_x` and `car1234_y` always share the same logical time. To ensure this property, it is preferable to localize such updates in a single module; e.g., the `Timer` class provided by SignalJ can be used for this purpose. More advanced language constructs to ensure that "both assignments occur simultaneously," such as parallel assignment (like `||` for parallel execution) can also be considered. To avoid introducing a new syntax specific to persistent signals, we introduced a new API method that makes the programmer's intention clearer, as shown here.

```
while (true) {
  car1234_x.set(getPosition().x).
    sync(car1234_y).set(getPosition().y);
  Thread.sleep(5000);
}
```

4.2 Time-Oriented Operations

Several query methods on persistent signals are defined. Those are grouped into three categories: basic selections, analytic queries, and domain specific queries.

4.2.1 Basic Selections

A basic selection filters a given persistent signal using a time condition; thus, the result of this operation is another signal that contains the time-series values that match with the specified time condition. This is similar to existing filter

Table 1. Persistent signal API (selective). In this table, we use T as a type parameter. For example, signal[T] is a type T whose modifier includes `signal`.

Type	Signature	Return type	Description
Basic	`within(java.sql.Timestamp ts, String interval)`	signal[T]	Time-series data within the extent specified by a time-stamp `ts` and an `interval` representing its interval
	`bucket(String interval)`	signal[T]	Time-series data using the sampling rate specified by `interval`
Analytic	`first()`	T	First value of the receiver signal
	`count()`	`int`	Number of records of the receiver signal
	`sum()`	T	Summation of records of the receiver signal
	`avg()`	T	Average of records of the receiver signal
	`max()`	T	Maximum value among records of the receiver signal
	`min()`	T	Minimum value among records of the receiver signal
	`histogram(T min, T max, int buckets)`	`T[]`	Histogram of the receiver signal with buckets of num. `buckets` defined over the range from `min` to `max`
Domain specific	`lastDiff(int i)`	T	Difference between the current value of the receiver signal and the `i`th value since the last value of that signal
	`distance(signal[T] s)`	T	Distance between the receiver signal and `s`

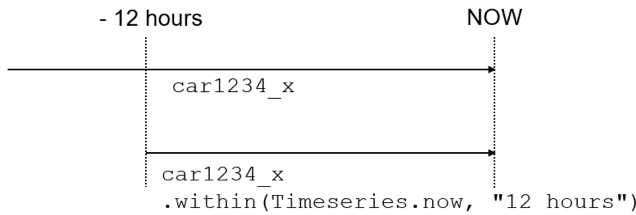

Figure 1. Selecting time-series data within the past 12 hours

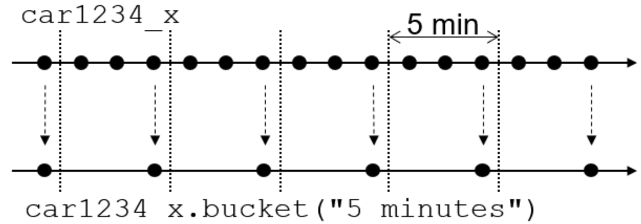

Figure 2. Changing sampling rate of time-series data

functions for signals and reactive streams. The difference is that our basic selection mechanisms provide the results based on time conditions. For example, to obtain all `car1234_x`'s values that have been recorded within the past 12 hours, we can use the `within` method (Figure 1).

```
signal int c12x =
    car1234_x.within(Timeseries.now, "12 hours");
```

This method takes two arguments: an instance of `java.sql.Timestamp` and a `String` value, which represents a time interval. `Timeseries` is a built-in class that contains useful information for time-oriented queries. For example, the field `now` holds an instance representing the "current time."[4] The `within` method returns a sequence of `car1234_x`'s time-series values restricted to only within the past 12 hours. Thus, the result of `within` is also a signal, and by accessing this signal (called `c12x` here), we can obtain the latest value of the sequence. We can additionally obtain the first value of the sequence using the `first` method.

```
int c12xfirst = c12x.first();
```

It may also be useful to obtain the time-series data with a coarser sampling rate. The `bucket` method is provided for this purpose (Figure 2). This method returns the time-series data using the sampling rate specified by the argument. For example, the following code results in two sequences of time-series x-coordinates and y-coordinates, respectively, for the above vehicle, sampled at a rate of every 5 minutes.

```
signal int car1234x5min =
    car1234_x.bucket("5 minutes");
signal int car1234y5min =
    car1234_y.bucket("5 minutes");
```

These time-oriented selection methods are applicable to any persistent signals, and the result should be considered a time-series data. Thus, we can chain these operations as follows:

[4]The field now holds the signal of `java.sql.Timestamp`, which means that this field returns the time at which it is accessed. Thus, `c12x` always holds the time-series data that have appeared in the past 12 hours prior to "now". If we need to creates a "fixed" time extent, we can use a non-signal instance of `java.sql.Timestamp`.

```
car1234_x.within(ts, "12 hours")
.bucket("5 minutes");
```

This example results in a sequence of car1234_x's time-series values limited to only those within the past 12 hours and sampled only every 5 minutes.

4.2.2 Analytic Queries

Persistent signals also support basic analytic queries such as count (the number of time-series values in the specified persistent signal), median (the median of the time-series values), avg (the average of the time-series values), sum (the sum of the time-series values), max (the maximum among the time-series values), and min (the minimum among the time-series values). Basically, those analytic functions return a non-signal value, but by declaring the return value as a signal, these results can also be handled as a signal. For example, the following signal count holds the number of c12x's time-series values, which varies when c12x is updated:

```
signal int count = c12x.count();
```

The result of analytic queries can also be a compound value. For example, the following histogram method returns a histogram of the receiver signal with eight buckets defined over the range from 10 to 90.

```
signal int[] h = c12x.histogram(10,90,8)
```

This histogram is represented as an array with ten buckets (including two more buckets other than the buckets specified in the argument for the histogram method: a bucket for values below the minimum threshold, and another for values above the maximum threshold). Because c12x is a signal and this histogram h is also declared as a signal, its value also changes over time.

4.2.3 Domain Specific Queries

Our language also supports a query API that is specific to some application domains. Our language processor consists of a compiler and a runtime library, and, as explained in Section 5, this runtime library is extensible, allowing it to be replaced with a library that supports a query API adapted to specific domains. This paper focuses on the domain of vehicle tracking. For example, in Table 1, lastDiff and distance are domain specific query methods that are designed for the vehicle tracking system.

Analytic queries can be applied to the view signals that are obtained by the domain specific queries. For example, in the following code fragment, max is called on v12, which is obtained by a call chain of domain specific queries.

```
signal int c12x =
    car1234_x.within(Timeseries.now, "12 hours");
signal int c12y =
    car1234_y.within(Timeseries.now, "12 hours");
signal int dx12 = c12x.lastDiff(1);
signal int dy12 = c12y.lastDiff(1);
signal int v12 = dx12.distance(dy12);
signal int v12max = v12.max();
```

How the domain specific queries are introduced is explained in Section 5.

4.2.4 Current Limitations

Because this work is in its early stage, the mechanism proposed here provides only preliminary features. Even though these features cover most basic features of existing time-series databases (such as TimescaleDB), it is preferable to extend the proposal to equip it with more convenient functions. For example, our proposal limits the base types of persistent signals to be primitives. This limitation makes it easy to implement the proposal and perform the feasibility study. However, in SignalJ, the underlying type of a signal can be any type allowed in Java, including reference types, and the ability to make such signals persistent would be helpful for programmers. We consider that a standard OR mapping mechanism may be suitable for allowing a compound type be treated as a persistent signal. Another possible extension will be designing language integrated queries based on persistent signals. Such queries are typically formed using a fluent API [9], and our API is also based on this idea. By enriching such API functions, we may construct more convenient queries.

5 Implementation

This section explains how the proposed mechanism is implemented using an existing time-series database system. To record all updates of persistent signals in the dedicated database system, we need to define mappings from the persistent signals to database instances. The fundamental policies of this mapping are as listed here.

- Each persistent signal is mapped into a key-value table, where the key is a timestamp indicating the time of the update, and the value contains the value of the signal at that time. The timestamp should be indexed according to the specific indexing mechanism used in the time-series database.
- Operations performed on a persistent signal are translated into the corresponding queries on the table that is mapped from the persistent signal.
- Each view signal is mapped to a view that uses the tables mapped from the persistent signals on which the view signal depends. Operations on view signals are translated into queries on the corresponding view.
- These mappings are kept consistent while the source code is evolving. A synchronization mechanism is provided to make the database schema consistent with the declarations of persistent signals in the source code.

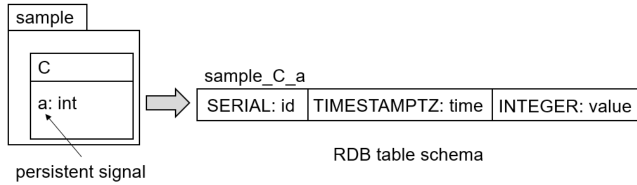

Figure 3. Mapping a persistent signal to an RDB table

- Because there are several time-series databases targeting different use case scenarios, it should be easy to replace the underlying DBMS with another one.

To achieve the last requirement, persistent signals are implemented based on the runtime library that contains the interface, namely, `PersistentSignal`, which provides the functionality to query the database. A concrete class implementing that interface provides the DBMS-specific mapping from the persistent signals into database instances. The runtime library is also responsible for providing domain specific queries.

This section presents an implementation based on TimescaleDB, a PostgreSQL-based time-series database. The reason that we use TimescaleDB as the backend for persistent signals is that it allows using the full power of SQL expressiveness in a way that makes it straightforward to implement the aforementioned mapping. Nevertheless, it is still an open question whether using TimescaleDB is the best solution among the currently available time-series databases. We will come back to this issue in Sections 6 and 7.

5.1 Key-Value Tables for Signals

Each persistent signal is implemented using a table that contains at least columns, namely, `time` and `value`, where `time` is the timestamp indicated when the update of the signal occurred and `value` is the value of the signal at that time.

Each signal also associates a serial number with each record. Figure 3 shows the mapping from a persistent signal to an RDB table. The name of the table is determined by concatenating the name of the signal and the qualified name of the class that declares the signal. The column `value` records every value held by that signal. Besides the timestamp, the column `id` (the serial number) is also used as a key of the table. For example, the table for `car1234_x` in the above example is created by the following `CREATE TABLE` statement (in the following, we omit the qualified class name for simplicity).

```
CREATE TABLE car1234_x (
  id    SERIAL,
  time  TIMESTAMPTZ NOT NULL,
  value INTEGER     NOT NULL
);
```

The difference between `time` and `id` is that, `time` is used to perform time-oriented queries as explained below, and `id` is used to identify records that were inserted at the *logically* same time. For example, in the vehicle tracking system explained in Section 4, updates to `car1234_x` and `car1234_y` occur simultaneously, but this fact is possibly not reflected by `time`. Thus, to perform queries on views that are derived from the tables (e.g., obtaining the velocity of the vehicle), we need to perform the natural-join of both tables by matching rows according to serial number.

The above table records time-series data, which have the following features.

- Each entry has a timestamp.
- Once inserted, entries are not normally updated.
- Recent entries are more likely to be queried at a fine-grained sampling rate (i.e., time interval)

To effectively interact with such time-series data, TimescaleDB provides an abstraction of a single continuous table across all space and time intervals; this is called a hypertable. All interactions with TimescaleDB (such as SQL queries) are implicitly with hypertables. A hypertable is created using the following `create_hypertable` function, which follows the `CREATE TABLE` statement.

```
SELECT create_hypertable('car1234_x', 'time');
```

The `CREATE TABLE` and `create_hypertable` statements, as well as the `CREATE VIEW` statements explained below, are not embedded in the bytecode generated by the compiler but are extracted as a separate SQL script. One issue is the execution timing of this script. We will return to this issue in Section 5.5.

5.2 Implementing Signal Operations using Database Queries

In the following sections, we explain how each access to a persistent signal is translated to the corresponding query for the database on a case-by-case basis. The first case is the update of a persistent signal using an assignment expression; for example, the following assignment inserts a new value to the corresponding table.

```
car1234_x = 88;
```

This assignment is translated to the following `INSERT INTO` statement, which inserts the pair of current timestamp and the assigned value (88 in this case) into the table representing the left-hand side of the assignment[5]:

```
INSERT INTO car1234_x(time, value)
  VALUES (NOW(), 88);
```

[5]To be precise, the compiler translates the assignment into a call to the runtime library method that executes the `INSERT INTO` statement. In the following, we explain the implementation on the basis of mapping to SQL statements for simplicity.

This statement also implicitly generates a new serial number, which is stored in the id column of the new record.

Each access to a persistent signal yields the most recent value of that signal, which can be implemented by the following SELECT statement.

```
SELECT value FROM car1234_x ORDER BY time
DESC LIMIT 1;
```

Actually, in SignalJ, the latest value is always cached in the object representing the signal, making a query unnecessary. In our implementation, this cache is always updated when a new value is inserted into the table. Thus, it is not necessary to issue the above query to obtain the latest value.

Time-oriented operations on persistent signals are implemented using the corresponding time-oriented queries provided by TimescaleDB. The following call of the within method is an example.

```
car1234_x.within(Timeseries.now, "12 hours");
```

This call is translated into the following SELECT statement.

```
SELECT * FROM car1234_x
WHERE time > NOW() - interval '12 hours';
```

In TimescaleDB, time-series data can be queried using the standard SELECT statement. This statement selects car1234_x's records that were inserted within the last 12 hours. The keyword interval is used to obtain the time extent represented using the timestamps. NOW() is equivalent to Timeseries.now, which always holds the current time. Thus, the result is also changing as time goes on.

One common case to use within is when we declare a view signal, such as c12x.

```
signal int c12x =
    car1234_x.within(Timeseries.now, "12 hours");
```

A declaration of a view signal is translated into a CREATE VIEW statement (that is extracted into the separate SQL script). Thus, in this case, the compiler generates the following CREATE VIEW statement.

```
CREATE VIEW c12 AS
  SELECT * FROM car1234_x
  WHERE time > NOW() - interval '12 hours';
```

We note that, as the value of NOW() is time-varying, this view also changes as time goes on.

Another example is creating a time bucket that contains the time-series data at a different sampling rate. The following signal records the update history of car1234_x using a 5-minute interval.

```
signal int car1234x5min =
    car1234_x.bucket("5 minutes");
```

This signal is implemented by the following CREATE VIEW statement using the time_bucket function of TimescaleDB.

```
CREATE VIEW car1234x5min AS
  SELECT DISTINCT
    id,
    time_bucket('5 minutes', time) AS time,
    last(value,time)
  FROM car1234_x
  GROUP BY time;
```

The time_bucket function truncates the timestamps using arbitrary time intervals. The last function, which is also provided by the TimescaleDB API, returns the latest value within the aggregated group.

Finally, analytic queries on the persistent (and view) signals are simply implemented using the analytic queries provided by SQL. For example, the expression car1234_x.count() is implemented using the following SELECT statement.

```
SELECT count(value) FROM car1234_x;
```

5.3 More Sophisticated Operations

Thanks to the expressiveness of SQL, more sophisticated operations on persistent signals can also be implemented. For example, the lastDiff method, which returns the difference between the current (i.e., latest) value of the signal and its specified recent past value, is represented by joining the same tables (views) using different offsets. We use the following example.

```
signal int dx12 = c12x.lastDiff(1);
```

This example uses lastDiff to declare the view signal dx12. This signal declaration is translated into the following CREATE VIEW statement, which uses the SELECT query derived from the call of lastDiff.

```
1 CREATE VIEW dx12 AS
2   SELECT
3     x.id AS id, x.value - y.value AS value
4   FROM
5     c12x AS x,
6     (SELECT id,value FROM c12x OFFSET 1) AS y
7   WHERE x.id = y.id - 1;
```

This statement first creates another view (line 6) that is the result of shifting the rows of c12x using the offset value specified as the argument to lastDiff (in this case, 1), and then joins this view with c12x using that offset (as indicated in the WHERE clause) and selects the difference between both values (line 3).

Operations that require multiple signals are also implemented in a similar way. The following signal v12 calculates the velocity of the vehicle using two signals, dx12 and dy12.

```
signal int v12 = dx12.distance(dy12);
```

This is straightforwardly implemented by creating a view by joining the views that correspond to those signals.

```
CREATE VIEW v12 AS
  SELECT
    dx12.id AS id,
    sqrt(dx12.value*dx12.value+
         dy12.value*dy12.value) AS value
  FROM dx12, dy12
  WHERE dx12.id = dy12.id
```

5.4 Notes on Glitches

Some RP languages introduce glitches, i.e., temporary inconsistencies in the signal networks. Thus, it is worth discussing whether this problem can occur in our approach.

SignalJ supports pull-based signals and push-based events. This means that a signal is re-evaluated whenever it is accessed (and thus it is guaranteed to be glitch-free), but events (i.e., an assignment to a source signal) are pushed to all composite signals that depend on the source signal. Because the event handler is called asynchronously, in general it can introduce glitches. From the viewpoint of persistent signals, an access to a view signal is pull-based; thus, the glitch freedom looks freely given. However, it is important to note that this glitch freedom is provided only when each record is associated with an appropriate serial number. The use of "parallel assignment" discussed above may synchronize all source signals in the same connected signal network, allowing glitches to be avoided.

5.5 Synchronizing Source Code with the Database

The above explanations of persistent signal semantics are based on SQL statements with an assumption that there is a database. Obviously, the database schema should be available before the application is running, and our language model hides the database details from the source code.

To achieve this requirement, we adopt an approach in which the database schema is automatically generated from the source code. The compiler analyzes the source code and determines the persistent and view signals that requires tables and views in the database schema. Then, the compiler runs a script that consists of the `CREATE` statements extracted from this analysis. This is similar to the model generation mechanism supported by Rails[6].

In taking such an approach, we must consider the software evolution. Usually, the software development process includes a sequence of compilations and test runs. In each test run, records stored in the database may be discarded. On the other hand, considering the software lifecycle, the application is likely to be modified after its release. While this modification may alter the database schema, it is preferable to migrate existing data into a database with the new schema, rather than clean the database contents. This schema alternation further raises an issue if the database is itself shared with other applications. Even though our proposal does not consider the database to be shared at first (the reason for using the database is its convenience for implementation of persistent signals), it is natural to consider exposing the data in the database to other applications.

At present, the `CREATE` statements are automatically generated from the source code. The remaining issue is when to execute the statements. Currently we assume that all persistent signals are static. That is, the compiler prepares all required tables and views. Executing `CREATE` statements at runtime will raise several issues regarding performance and garbage collection. We leave solving these issues as future work.

To execute all these features, we assume that the developer first selects the *compilation mode*, indicating whether the database contents should be preserved after the compilation, and/or whether they will be shared with other applications. The compiler then generates an SQL script that, according to this mode, either cleans up the tables and views and creates new ones or alters the database schema, and builds the whole application. This process is further explained below for each compilation mode.

- **Clean-up Mode:** In this mode, all tables and views are dropped and new ones are created during compilation. This mode is selected when retaining the existing database contents is not necessary. For example, records in the database are stored by test runs. In this case, those records are not necessary in the subsequent modifications.
- **Data Preserving Mode:** In this mode, the compiler preserves the database contents and alters the tables and views when necessary. The compiler analyzes whether the modification requires any changes in the database schema by comparing the database schema generated from the modified source code with the existing schema. If these are identical, there is no need to change the database schema and the compiler does not execute any SQL statements. If there are differences, the compiler drops tables for all removed persistent signals, creates tables for all newly introduced persistent signals, alters tables and views for all persistent signals whose definitions have been altered, and drops and creates views for all affected views. This mode is selected in the case where the database contains data used in the running application.
- **Data Sharing Mode:** In this mode, the compiler compares the database schema generated from the modified source code with the existing schema. If there are differences, the compiler does not change the existing schema but generates a mapper from the new schema to the existing schema if possible. For example, we can generate such a mapper when some signals have been renamed. In other cases, the compiler reports an error.

[6]http://rubyonrails.org

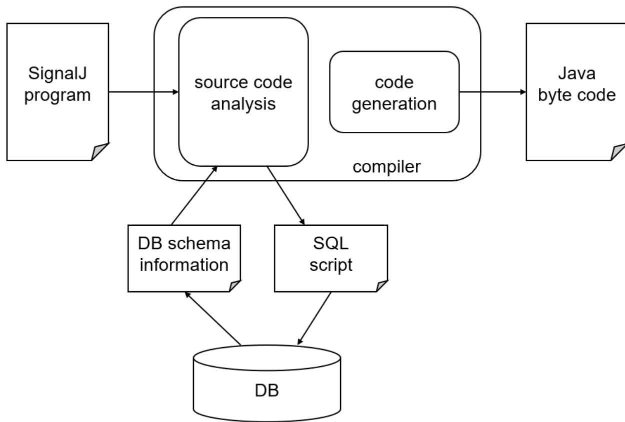

Figure 4. Compilation process

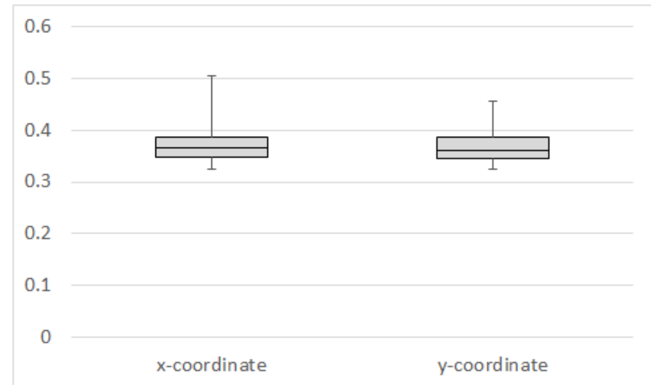

Figure 5. Response time of the `within` query (ms)

To be accurate, the compiler should be aware of the intention of the source code modification. For example, in the data preserving mode, the compiler should not drop the table in cases where the persistent signal has merely been renamed. One approach to letting the compiler know the intention is to provide an interactive interface that enables the programmer to communicate with the compiler. In this paper, we do not consider this issue further.

Figure 4 summarizes this process. Our compiler reads and analyzes the SignalJ program. This analysis includes comparison between the current database schema and signal declarations, and the compiler generates the SQL script to modify the database schema. The database subsystem runs this script to update the database schema (this implies that the database subsystem should be available during compilation). Meanwhile, the compiler generates Java bytecode.

6 Preliminary Evaluation

The proposed language relies on the mapping from persistent signals to the underlying database instance. To simply explain the semantics of persistent signals using the underlying database mechanisms, this mapping is kept simple. Before considering a more efficient mapping, it is worth studying how feasible our approach is by measuring its performance in this setting. For this purpose, we conducted preliminary microbenchmark experiments that measure the response time for each time-oriented query, analytic query, and insertion of new records into the tables that represent the persistent signals.

These microbenchmarks were performed on Linux kernel version 3.10.0 running on a four-cores Intel Core i5-4440 3.10GHz CPU with 8 GB of main memory. The underlying database management system was PostgreSQL 9.6.0 with TimescaleDB 1.1.1. For this benchmarking, we tested the aforementioned vehicle tracking example. We created 100 vehicles (i.e., 200 signals (tables) that correspond to the x- and y-coordinates of each vehicle). We also created five views for each vehicle: views that hold the last 12 hours of data for each coordinate (12x and 12y, resp.), views that hold the difference between values in 12x and 12y and their previous values (12dx and 12dy, resp.), and a view that holds the estimated velocities of the vehicle during the last 12 hours (12v). In short, as examples in Sections 4 and 5, these views were created using `within`. Each table consists of 22,032 records inserted at 10 minute intervals over three days (resulting in 432 records), at 1 minute intervals over three days (resulting in 4,320 records), and at 5 second intervals over one day (resulting in 17,280 records). The size of the database was 414 MB in total.

Our purpose is not to find the best database configurations that gives the best performance. Here, we are judging the feasibility of the proposed system, so all experiments were conducted under the default settings, using a small shared buffer (128 MB) and enabling the autocommit feature. We aimed for a response time of less than 10 ms, which will be satisfactory performance for many applications.

We did not compare between versions of SignalJ with and without persistent signals by running the program without persistent signals because the extended SignalJ compiler actually generates the same code whether the source code contains any persistent signals or not.

6.1 Result 1: Time-Oriented Queries

First, we measured the performance of the time-oriented analytic query (i.e., `within`) because we assume it will be intensively used, in particular for creating view signals. This measurement was performed by issuing the following query for each table (thus, 200 queries were executed in total).

```
SELECT value FROM [table name]
  WHERE time > NOW() - interval '10 minutes'
```

This query returns all records inserted within the last ten minutes. The box-and-whisker plot of the response times is shown in Figure 5. The average response time was 0.370

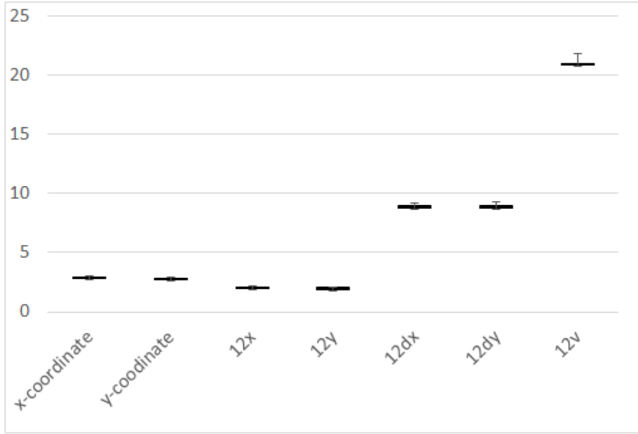

Figure 6. Response time of the avg query (ms)

Table 2. Average of response time of the avg query (ms)

x-coord.	y-coord.	12x	12y	12dx	12dy	12v
2.83	2.70	1.99	1.93	8.87	8.87	21.0

ms. The figure shows that most queries were performed within less than 0.4 ms, and the worst case was approximately 0.5 ms. This result shows that TimescaleDB, which adds hooks into PostgreSQL's query planner, data model, and execution engine to enable high-performance interactions with time-series data, is quite responsive and stable for the time-oriented queries used in the proposed language.

6.2 Result 2: Analytic Queries

We also measured the performance of other analytic queries such as `count`, `sum`, `avg`, and `max`. We executed these queries against each table and view. The results are shown in Figure 6, which shows the case of `avg`. The average response time is shown in Table 2. The results of other analytic queries were very similar. These results indicate that, compared with `within`, the analytic queries require more response time. Nevertheless, all queries (except for those performed on 12v) were performed within less than 10 ms, and their performance was stable in that there were no outliers. Queries performed on the views of 12v were quite slow because these views were constructed using a complex `SELECT` subquery to calculate the velocity of the vehicle.

Interestingly, the response times for views of 12x and 12y were a bit faster than those for tables. We consider that this is because the size of each view (which holds the time-series values that were inserted only within the last 12 hours) is smaller than that of the underlying table (which holds all past time-series data). Moreover, the `SELECT` queries used in the `CREATE VIEW` statements rely on time-oriented queries, which were shown to be quite fast in Section 6.1. Thus, even though those views were created at runtime, their constructions were fast enough to overcome the longer response time of the analytic queries. We will also see this tendency in Section 6.4.

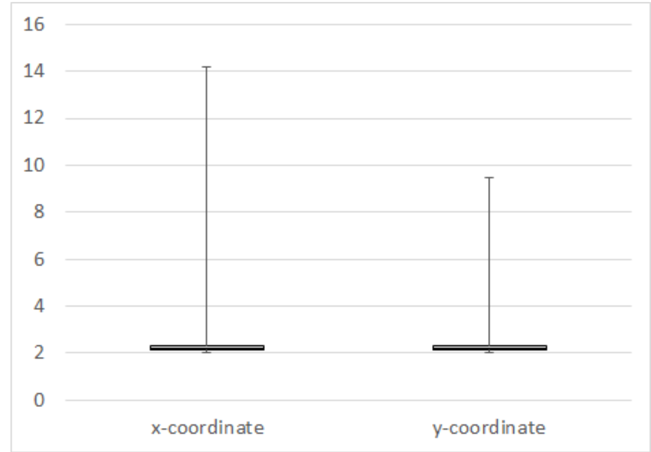

Figure 7. Response time when assigning a new value to a signal (ms)

6.3 Result 3: Appending Data

To measure the response time when assigning a new value to a signal, we executed the `INSERT` query, which inserts a new value to the signal for each table. The result is shown in Figure 7. This figure indicates that most of the data insertion required nearly 2 ms and thus the results were mostly satisfactory. However, compared with the query results, the performance of data insertion looks unstable. Some of the insertions required a much longer response time, and the worst case was over 14 ms. Thus, there might be cases in which the latest value has not yet been inserted to the table at the time when the corresponding persistent signal is accessed. The caching mechanism that holds the most recent values in the main memory will compensate for this problem.

6.4 Results with Increasing Number of Records

We also studied how the number of records in the database affects the performance by measuring the query response times of tables and views with different numbers of records in the tables. This experiment was performed in tables with 100, 1,000, 10,000, and 100,000 records. The size of each database was 28 MB, 50 MB, 205 MB, and 1.76 GB, respectively. These records were inserted at a rate of one every 5 seconds.

Figure 8 shows the average response times of `within` measured by issuing the queries shown in Section 6.1 for each number of records. This graph indicates that the number of records does not significantly affect performance. Even when each table contains of 100,000 records, most queries were completed within less than 0.4 ms, which is similar to the result shown in Figure 5.

Figure 9 shows the average response times of `avg` for each number of records. This graph also shows the results

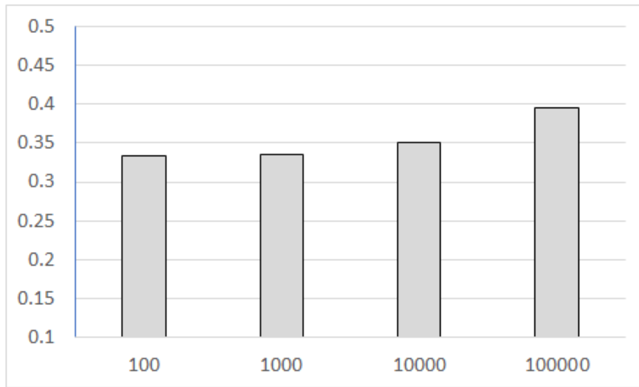

Figure 8. Response time of `within` with 100, 1,000, 10,000, and 100,000 records (ms)

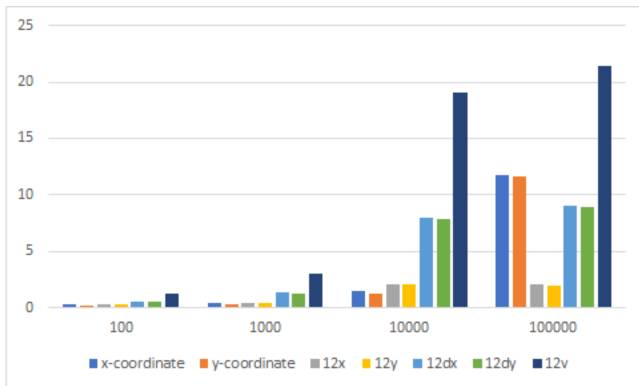

Figure 9. Response time of `avg` with 100, 1,000, 10,000, and 100,000 records (ms)

for views derived from the tables (12x, 12y, 12dx, 12dy, and 12v). This graph indicates that although queries on tables took longer when the number of records was higher, the time needed for queries on views 12x and 12y was not significantly affected by the number of records. We can also see this tendency in views 12dx, 12dy, and 12v (which are constructed using complex `SELECT` subqueries) when the number of records is large. Queries performed on 12dx and 12dy were faster than those performed on the underlying tables when each table consists of 100,000 records. We also tested the same query on the database with tables containing 1,000,000 records (the size was 17 GB), and observed that the response times of queries performed on views did not increase. This is consistent with the results shown in Section 6.1: because each view contains a limited number of records and view construction is quite fast, the response time of `avg` on a view is insulated from the influence of the number of records in the tables.

These results indicate that time-oriented queries are quite useful for filtering out unnecessary data before performing analytic queries when the number of time-series records is large.

6.5 Summary

Taken together, the above experimental results show that our preliminary implementation is responsive in most cases. Although queries performed on a view constructed by a complex `SELECT` subquery were slow, other queries could be performed within our desired response time. Thus, we conclude that our approach to persistent signals will be feasible for many applications, and additional study will further enhance the performance.

We note some threats to validity of this study. First, we did not perform any stress tests in which a large number of queries were issued simultaneously. In general, stress testing requires a practical environment in which a real service will be deployed. In such an environment, the database should be configured so as to make it fault tolerant and high performance. Unfortunately, we could not prepare such an environment; the experiments described in this paper were conducted on a standalone database running on a single desktop PC without redundancy or distribution. Nevertheless, we are optimistic about making it possible for the persistent signals to handle a large amount of queries simultaneously. The results of other research indicate that the existing time-series databases are scalable enough [13].

Because the above experiments were intended as a preliminary evaluation to judge whether it is valuable to continue study in this direction, we did not apply any optimizations to the database, conducting the experiments under the default settings. Altering the configuration by changing the memory settings, the number of worker threads, and lock management style would yield different results. Furthermore, we may also consider another DBMS because the ACID properties provided by an RDBMS may be less relevant for persistent signals. If the ACID properties are relaxed, NoSQL-based time-series databases become candidates, although there is still a tradeoff between performance and expressiveness. Finding an ideal DBMS implementation for our purpose remains as future work.

7 Related Work

The techniques for RP have been inspired by synchronous languages [3, 10, 23] and functional-reactive programming (FRP) languages [7]. FRP features have been embedded in general-purpose functional languages (e.g., the Yampa library [20] for Haskell), and recently such features have made their way into imperative object-oriented settings [14, 18, 24] by integrating signals with event-based programming features [8].

Debugging for RP is enhanced by time-traveling [21], which records the execution history of the application to make it possible to pause the execution and rewind to any

earlier execution point. Time-traveling is now common in RP debuggers. For example, Reactive Inspector [25], a debugger for REScala [24], visualizes how signal networks are constructed and evolved and how propagations take place over those networks during execution. Using this debugger, a programmer can see the status of the networks at any execution point. In other words, these tools stores time-series values for each signal to make time-traveling possible. In this sense, these tools are related to the system proposed in this paper. Some tools also provide visualization of such time-series data, such as allowing viewing of the execution history in a single display to identify anomaly propagation patterns that are repeated over time [2, 11, 12]. Usually, such tools are dedicated to debugging, and the time-series data handled in them are not provided for use by applications. For example, no convenient APIs to query over such time-series data are provided.

In our study, persistent signals were implemented using TimescaleDB, a time-series extension to PostgreSQL. Jensen et al. presented a survey on time-series databases, which are also known as time-series management systems [13]. In their survey, time-series databases were categorized into internal data stores, external data stores, and RDBMS extensions. An internal data store (e.g., *tsdb* presented by Deri et al. [5]) integrates both a data store and a processing engine together in the same application, allowing for deep integration between the storage and the processing engine. In another approach, an external data store (e.g., *Gorilla* by Pelkonen et al. [22] and *BTrDB* by Andersen and Culler [1]) uses an existing external DBMS, allowing for existing DBMS deployments to be reused. Finally, an RDBMS extension (e.g., *TimeTravel* by Khalefa et al. [15]) allows the expressive power of the RDB to be applied to the time-series database. TimescaleDB, used in our implementation, falls into this last category. Overall, there have been many time-series database implementations suitable for different use case scenarios. Although the experimental results shown in this paper indicate that using TimescaleDB might be satisfactory in many cases, it will be beneficial to consider other implementations that are specialized to our domain. This is left to future work.

In the proposed language, the programmers do not construct a database query explicitly but a programming interface is provided to perform a query. In this sense, our approach is based on (a limited version of) language integrated queries. One well-known implementation of language integrated queries is LINQ [17], which is available for the .NET framework. ScalaQL [27] is another example, implemented on Scala. jOOQ[7], a Java-based query interface, provides language integrated queries in the form of a fluent API. Relative to embedding SQL statements in the host languages, language integrated queries are preferred because they reduce the impedance mismatch between the application and the database layers and enable the compiler to check the type safety of the queries. Our approach also adopts this, integrating the time-oriented queries into the language even though the available API is still limited. Designing more expressive language integrated time-oriented queries remains as future work.

[7] http://www.jooq.org

8 Conclusions and Future Work

This paper proposed an approach for persistent signals that bridges the gap between RP and time-series databases. For this purpose, SignalJ, a Java-based RP language, is extended to allow explicit declaration of persistent signals and to provide an API for performing time-oriented and analytic queries on such signals. A preliminary implementation based on TimescaleDB, a PostgreSQL-based time-series database, was developed, and simple microbenchmarks of this system showed that the proposed approach is promising in most respects.

This work also clearifies a variety of paths for future work, as listed below.

8.1 Performance

Pursuing the best tuning for the DBMS will provide better performance. Furthermore, we may also consider another implementation based on a different DBMS. In doing so, we should consider the tradeoff between the performance and expressiveness. Even though the current implementation is realized using an external DBMS, this setting is not inherently necessary, and we may also consider an internal data store that is specialized to our domain. Thus, finding an ideal implementation will be a valuable subject for study in the future.

8.2 Non-primitive Persistent Signals

Another direction for future research is to extend the API and the data model. Currently, the persistent signals support only primitive types. However, in SignalJ, any objects can be declared as a signal. To make such signals persistent, we need to develop a mapping from the object to records in a database and provide a method to make them time-series data. Existing OR mapping mechanisms might be usable for this, but we need to be careful to ensure consistency without sacrificing performance. Thus, this direction of research is also worth considering.

8.3 Dynamically Generated Persistent Signals

One significant limitation in our approach is that each persistent signal should be statically declared. This means that in the microbenchmark, each car `carXXX` corresponds to one of 100 hard-coded vehicles. This is unrealistic in many IoT applications. Instead, we could dynamically generate a signal representing the vehicle that need to be inspected.

The above mentioned non-primitive persistent signals may address this problem. Assuming that carTable is a persistent signal that holds a join of all vehicles, we could declare a signal carX that represents one of the cars.

```
persistent signal Vehicles carTable = ...
signal int carX =
    carTable.select(/* some query keys*/);
signal int c12x =
    carX.within(Timeseries.now, "12 hours");
```

In this case, carX is a view signal on which we can perform any time-oriented queries. Assuming that the query key is also declared as a signal, its change will be propagated to carX, resulting in a new instance being referenced by carX. This "switch" will be further propagated to c12x.

In this scenario, each instance referenced by carX corresponds to a view in the database. Obviously, this requires dynamic construction of views. We also need to consider garbage collection when a view becomes inaccessible from the program, with the caveat that we might want to reuse a view that becomes accessible again (such as by issuing a query that was issued earlier).

We note that carTable is also a (persistent) signal. Unlike the above scenario, a change in carTable should not result in the switch of the instance referred to by carX. This means that we need to develop a different propagation strategy (not switching the persistent signal but instead changing its internal time-series data) for the network of persistent signals whose records are non-primitive.

In summary, non-primitive persistent signals and dynamic construction of persistent signals are related issues that are worth considering.

8.4 Migration

This paper does not consider the scenario in which a time-series database already exists and a new application is to be developed using signals and the database. Because time-series databases handle non-primitive values in general, the above discussed OR mapping will be useful in such a case. One issue will be the handling of time, because most of the existing time-series databases do not provide serial numbers representing the logical time. To supply the logical time, we need to identify which entries are considered synchronized, and this migration to argmented time-series data should be automated. There are many issues (e.g., automatic detection of synchronized data) worth considering in future work.

8.5 Distribution and Glitches

Finally, designing efficient update propagation with consistency guarantees in distributed RP has been identified as an interesting problem [6, 16, 19]. This issue becomes even more interesting when we consider transactions, that is, handling of synchronizied signals (i.e., signals with the same logical time). We are planning to apply recently developed distributed database mechanisms[26, 28] that provide scale-out capabilities while ensuring the data consistency to the implementation of persistent signals. A PostgreSQL-based distributed database system is also available[8]. This research will be performed under the assumption that the semantics of a distributed RP can be explained using distributed database mechanisms.

Because computation of signals in SignalJ is basically pull-based, it is inherently glitch-free, and this property will also be held in distributed settings. However, SignalJ does provides an asynchronous push-based event mechanism, and for some uses push-based computation of signals is necessary (e.g., the aforementioned dynamic construction of view signals). Thus, glitches are still potential issues in our setting, and a scheduling mechanism to avoid them is worth considering.

Acknowledgements. This research was supported by the Kayamori Foundation of Informational Science Advancement and JSPS KAKENHI Grant Number 17K00115.

References

[1] Michael P. Andersen and David E. Culler. BTrDB: Optimizing storage system design for timeseries processing. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 39–52, 2016.

[2] Herman Banken, Erik Meijer, and Georgious Gousios. Debugging data flows in reactive programs. In *ICSE'18*, pages 752–763, 2018.

[3] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] Gregory H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language.* PhD thesis, Department of Computer Science, Brown University, 2008.

[5] Luca Deri, Simone Mainardi, and Francesco Fusco. tsdb: A compressed database for time series. In *TMA*, 2012.

[6] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: an update algorithm for distributed reactive programming. In *OOPSLA'14*, pages 361–376, 2014.

[7] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 263–273, 1997.

[8] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: modular event-driven object interactions in Scala. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD'11)*, pages 227–240, 2011.

[9] Yossi Gil and Tomer Levy. Formal language recognition with the Java type checker. In *ECOOP 2016*, pages 10:1–10:27, 2016.

[10] Nicholas Halbwachs, Paul Caspi, Pascal Paymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[11] Takumi Hikosaka, Tetsuo Kamina, and Katsuhisa Maruyama. Visualizing reactive execution history using propagation traces. In *REBLS'18*, 2018.

[12] Jeff Horemans and Bob Reynders. Elmsvuur: A multi-tier version of elm and its time-traveling debugger. In *TFP 2017*, volume 10788 of *LNCS*, pages 79–97, 2017.

[8] https://www.citusdata.com/

[13] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 29:2581–2600, 2018.

[14] Tetsuo Kamina and Tomoyuki Aotani. Harmonizing signals and events with a lightweight extension to Java. *The Art, Science, and Engineering of Programming*, 2(3), 2018.

[15] Mohamed E. Khalefa, Ulrike Fischer, Torben Bach Pedersen, and Wolfgang Lehner. Model-based integration of past & future in TimeTravel. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 1974–1977, 2012.

[16] Alessandro Margara and Guido Salvaneschi. We have a DREAM: distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems (DEBS'14)*, pages 142–153, 2014.

[17] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *ICMD 2006*, 2006.

[18] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Application (OOPSLA'09)*, pages 1–20, 2009.

[19] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Distributed reactive programming for reactive distributed systems. *The Art, Science, and Engineering of Programming*, 3(3):5:1–5:52, 2019.

[20] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell'02)*, pages 51–64, 2002.

[21] Laszlo Pandy. Bret Victor style reactive debugging. Elm Workshop, 2013.

[22] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, 2015.

[23] Marc Pouzet. *Lucid Synchrone version 3.0: Tutorial and Reference Manual*. Université Paris-Sud, LRI, April 2006. Online manual.

[24] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY'14)*, pages 25–36, 2014.

[25] Guido Salvaneschi and Mira Mezini. Debugging for reactive programming. In *ICSE'16*, pages 796–807, 2016.

[26] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[27] Daniel Spiewak and Tian Zhao. ScalaQL: Language-integrated database queries for Scala. In *SLE 2009*, pages 154–163, 2009.

[28] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD'17*, pages 1041–1052, 2017.

[29] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *PADL 2002: Practical Aspects of Declarative Languages*, volume 2257 of *LNCS*, pages 155–172, 2002.