UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

A Comparison of Stream Evaluations: TeSSLa versus stream-based Databases

Vergleich der Datenstromauswertung: TeSSLa und strombasierte Datenbanken

Bachelorarbeit

verfasst am
Institut für Softwaretechnik und Programmiersprachen

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Tim Schulz

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Assoc. Prof. Volker Stolz

Lübeck, den 9. Juni 2020

IM FOCUS DAS LEBEN

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Tim Schulz

Zusammenfassung

Stromdatenbanken sind Systeme zur Anfragebeantwortung auf Datenströmen. Einige davon basieren auf Ähnlichkeiten zu klassischen Datenbanken und sind daher als Erweiterung solcher umgesetzt. Wie in der Einleitung noch genauer erörtert, nennen wir diese Systeme, als die eine Seite des Vergleichs, „strombasierte Datenbanken “. TeSSLa (Temporal Stream-based Specification Language) auf der anderen Seite ist nicht direkt eine Stromdatenbank, sondern mehr, wie der Name schon andeutet, eine Sprache zur Formulierung von Datenströmen, für die es jedoch Implementierungen gibt, die anhand einer TeSSLa Spezifikation konkrete Datenströme verarbeiten können. Der Vergleich findet also zwischen strombasierten Datenbanken mit ihren Anfragesprachen und TeSSLa mit ihrer Anwendung statt.

Da die strombasierten Datenbanken, die wir betrachten, für einen allgemeinen Einsatz auf Datenströmen mit hohem Zieldurchsatz konzipiert sind, werden wir annehmen, dass sie prinzipiell auch für Datenstromverarbeitung in typischen Anwendungsbereichen von TeSSLa eingesetzt werden können. Um dieses Konzept zu veranschaulichen, werden wir eine einfache Implementierung vorstellen, die eine Erweiterung von PostgreSQL namens PipelineDB verwendet, um typische Beispiele für TeSSLa Anwendungen umzusetzen. Einen Vergleich zur Performanz für typische Stream Runtime Verification Anwendungen werden wir jedoch nicht führen.

Anschließend stellen wir eine Implementierung des Linear Road Benchmarks, der als typisches Anwendungsbeispiel für Stromdatenbanken entworfen wurde, mit TeSSLa vor. Damit werden wir demonstrieren, dass sich entsprechende Anwendungsfälle in TeSSLa formulieren lassen, die derzeitige Implementierung des TeSSLa Interpreters bei umfangreichem Gebrauch von unbegrenzten Datenstrukturen anderen Stromdatenbanken in der Performanz nachsteht. Mit dem Transcompiler von TeSSLa zu Scala erhalten wir jedoch eine effiziente Lösung, mit der wir eine Simulation der Skalierung von 10 Expressways erfolgreich bearbeiten.

Vor den konkreten Implementierungen werden wir TeSSLa mit Anfragesprachen für strombasierte Datenbanken vergleichen, um weitere Einsichten zu Vor- und Nachteilen der jeweiligen Systeme bei verschiedenen Anwendungen darzulegen.

Abstract

Data Stream Management Systems (DSMS) are built to manage queries on data streams. Some of these attempts are based on similarities to classical databases and therefore enhance relational databases. As we will see in the introduction, this is what we will consider a stream-based database (SBD) in our comparison. TeSSLa (Temporal Stream-based Specification Language) itself on the other hand is not a DSMS, rather it is a specification language which comes with tool chains for using a TeSSLa specification to actually evaluate data streams. Therefore we will compare SBD with their query language and performance to TeSSLa specifications and their application.

Since the SBDs we consider are designed for a general use and aiming for high throughput, we will assume, that they might be used for the same tasks as TeSSLa is. To demonstrate this concept we will use a simple implementation which utilizes PipelineDB, an extension to PostgreSQL, to perform on small standard examples for TeSSLa's applications. The performance comparison to TeSSLa on typical Stream Runtime Verification applications will remain open.

The next step will be to implement the Linear Road Benchmark, which was originally designed as a typical use-case for DSMS, with TeSSLa. By that we will show that the language can be used to formulate such problems indeed, but the performance evaluation will suggest, that the current TeSSLa-assembly implementation can, by massively using unbounded data structures, only handle a comparatively small workload while meeting the latency requirements set by the benchmark. However the TeSSLa-to-Scala compiler yields an efficient solution, capable of handling full 10 expressways.

Before the practical implementations however, we will compare specification features of TeSSLa and SBD query languages to give further hints on advantages and disadvantages of these systems for different tasks.

Contents

1 Introduction 1

1.1 Contributions of this Thesis 2
1.2 Related Work 2
1.3 Structure of this Thesis 3

2 Specification and Query Languages 4

2.1 TeSSLa 4
2.2 CQL 7
2.3 SQL for PipelineDB 9

3 Example Use Cases 13

3.1 Linear Road Benchmark 13
3.2 Burst Pattern 19

4 Stream Property Implementations 22

4.1 Linear Road: Daily Expenditures 22
4.2 Linear Road: Accident Alerts 27
4.3 Burst Pattern Example 43
4.4 Property Implementation Summary 52

5 SRV Implementation with PipelineDB 55

5.1 Implemented Program Structure 55
5.2 SQL Specification File 56
5.3 SRV PipelineDB Summary 58

6 Linear Road Implementation with TeSSLa 60

6.1 Program Structure 60
6.2 Notes on Correctness 61
6.3 Simulation Data 62
6.4 Performance Examination 63

7 Conclusion 71

Bibliography 72

1

Introduction

Processing data streams, which we will assume in this work to be discrete, is still an open topic meaning there is no standard in specifying data streams and formulating queries on them yet. The CQL Continuous Query language [1] is an attempt to enhance a relational query language like SQL by operators producing relations out of data streams and vice versa. With various relation-to-relation operators of the underlying query language this yields a dual approach in which a query starting from a stream and outputting a stream might take the detour over a relation like some window function and turning this (shifting) relation back into a stream. In chapter 10 of [1] they compare their language to other attempts and, in particular, show their rich expressiveness in comparison to stream-only attempts. They state that "the concept of a relation is useful even in applications whose inputs and outputs are all streams." Later on, we will use CQL for some reference specifications for SBD in comparison to TeSSLa since they use the relational approach and make a clear distinction between streams and relations. The Stanford Data Stream Management System [3] is an implementation of a general purpose DSMS using CQL, however we will not use it for our implementation since it is just a prototype and does not fully support the language. Instead we will use PipelineDB, which is an extension to PostgreSQL, treating data streams as append-only tables.

On the other hand there are approaches to stream processing focusing on less general purposes opening up room for performance improvements: TeSSLa is designed in the context of Stream Runtime Verification [6]. SRV is a technique using a program trace as an input data stream and incrementally deriving output streams allowing to check, whether the observed system is running correctly. In this program trace, the events are, in particular, ordered by their timestamp. This implies a close relation to Time Series Databases (TSDB) which are built to store and work on such series, for example to discover motifs as described in [12]. While sometimes time series are defined as a sequence of explicit length n that is completely known to the system [12], which would define a TSDB as a specialized DBMS with time as a key field, other authors also emphasize the challenges on time series as input streams [22, 17, 23], which therefore describes TSDBs as specialized DSMSs with time as special key field. In this sense one can say, that the comparison between TeSSLa and SBDs is a comparison between a particular TSDB and general DSMSs with a relational approach.

The approach taken for representing the data in TeSSLa is stream-only and, using

basic data types, each of the basic operators can be evaluated with constant time and memory requirements. Thus there are some easy to check criteria, implying that a specification can be run with constant time and memory bounds assuring that critical real time constraints can be met even on massive data streams. In particular they proved in [6] that TeSSLa_{bool+c}, the fragment of TeSSLa restricted to checking event ordering and comparing timestamps to constants, is *PSPACE*-complete. Further, if a specification happens to feature constant resource requirements, the specified behavior can even be implemented in hardware like FPGAs [5]. Despite the focus of SRV, TeSSLa is still designed for general purpose stream specifications, therefore the question arises how this approach performs on different streaming data use-cases in comparison to approaches introducing relations – in terms of specifying the behavior on input streams as well as the actual performance running on them.

1.1 Contributions of this Thesis

In this work we will compare TeSSLa specifications of exemplary use cases with implementations in SQL for PipelineDB and CQL. Since TeSSLa on the one hand treats time as a "First-Class Citizen" [6] and our SBDs on the other hand have a close relation to conventional DBMSs, the comparison will focus on special timing criteria as well as challenges of storing and working on a mass of stored data. Further we will utilize PipelineDB for simple SRV tasks and implement the Linear Road Benchmark [2] using TeSSLa to compare TeSSLa's compiler and interpreter's performance as a general purpose Data Stream Management System to other available systems.

1.2 Related Work

There have been other DSMS taking the challenge of the linear road benchmark: First of all, in the original paper of the benchmark [2], they describe implementations using a commercially available Relational Database and the DSMS Aurora. Later on, other implementations like in the MaxStream project [4] or using Apache Flink [8] or Streamonas [11] were able to achieve decent performance. Also, besides Linear Road, there are other benchmarks for streaming applications like [25]. In [1] CQL is compared to other languages including its expressiveness in comparison to stream-only languages. Limitations of relational algebra and SQL as well as enrichments regarding streams have been studied in [10]. In particular, they talk about the loss of expressiveness one will face by banning blocking operators. As an approach to counter this issue, they name User Defined Aggregates (UDAs) which, in their examples, are defined in a procedural way enabling to precess the tuples in the order they appear and thus, in a way, taking a step back from purely relational languages and moving more in the direction of languages directly designed for data streams. TeSSLa has briefly been compared to other stream-only specification languages [6] but not yet to query languages of relational DSMS and further there has not been an implementation of a general purpose Data Stream Management System benchmark with TeSSLa before.

1.3 Structure of this Thesis

In the following chapters we will first provide a brief overview on the specification languages we will consider, namely TeSSLa, CQL and SQL for PipelineDB. Then we will outline two use cases, the setting of Linear Road and the Burst Pattern on example streams, to further pick some of these properties for demonstrating and comparing implementations in the different languages. After that, we will show how PipelineDB could be utilized for SRV in a similar manner one can use the TeSSLa compiler and interpreter on data in TeSSLa's input stream format. In the end, we will introduce the Linear Road Benchmark implementation with TeSSLa and discuss the results.

2

Specification and Query Languages

Since we aim for a comparison between the following three languages, we will present all of them following the same concept: First we will sketch fundamental ideas then present basic operators and finally give some examples. We will only present a brief overview, hence one might read the original papers and documentations for more detailed information.

2.1 TeSSLa

In the Temporal Stream-based Specification language [6] streams consist of timed events of certain domains i.e. a stream features a time and a data domain whereas every event in this stream consists of a unique time from the time domain and some value from the data domain. Definition 1 of [6] limits the time domain to a totally ordered semi-ring $(\mathbb{T}, 0, 1, +, \cdot, \leq)$ with $\forall t \in \mathbb{T} : 0 \leq t$. A TeSSLa specification can use multiple input streams and produce multiple output streams, where the arrival of the input events must be globally ordered by their time stamps.

Basic Operators and Constants

Let $\mathcal{S}_{\mathbb{D}}$ be the set of streams over some data domain $\mathbb{D}$. Let further $\mathbb{U} = \{\Box\}$ be the unit domain, used whenever the domain only contains one element and $\mathbb{T}$ be the time domain.

Nil The constant **nil** denotes the stream in $\mathcal{S}_{\mathbb{D}}$ containing not a single event.

Unit The constant **unit** denotes the stream in $\mathcal{S}_{\mathbb{U}}$ over the unit domain $\mathbb{U}$ with just a single event at time 0.

Time The **time** operator is of type $\mathcal{S}_{\mathbb{D}} \rightarrow \mathcal{S}_{\mathbb{T}}$. Given a stream s it produces the stream of timestamps, replacing every event's value in s with the corresponding timestamp.

Unary lift The $\mathbf{lift_1}$ operator of the type $(\mathbb{D} \rightarrow \mathbb{D}') \rightarrow (\mathcal{S}_{\mathbb{D}} \rightarrow \mathcal{S}_{\mathbb{D}'})$ with a function f of the type $\mathbb{D} \rightarrow \mathbb{D}'$ operates on a stream s of the type $\mathcal{S}_{\mathbb{D}}$ by applying f on the value of each event in s, producing the corresponding stream $s' \in \mathcal{S}_{\mathbb{D}'}$.

Binary lift The $\mathbf{lift_2}$ operator of the type $(\mathbb{D}_1 \times \mathbb{D}_2 \rightarrow \mathbb{D}') \rightarrow (\mathcal{S}_{\mathbb{D}_1} \times \mathcal{S}_{\mathbb{D}_2} \rightarrow \mathcal{S}_{\mathbb{D}'})$ with a function f of the type $\mathbb{D}_1 \times \mathbb{D}_2 \rightarrow \mathbb{D}'$ operates on a pair of stream s_1, s_2 of the types

$\mathcal{S}_{\mathbb{D}_1}, \mathcal{S}_{\mathbb{D}_2}$ by applying f on the values of the events in s_1 and s_2, producing the corresponding stream $s' \in \mathcal{S}_{\mathbb{D}'}$. Note that the binary lift can be used to define an n-ary lift.

Last The **last** operator of the type $\mathcal{S}_{\mathbb{D}} \times \mathcal{S}_{\mathbb{D}'}$ operates on a pair of stream s_1, s_2 of the types $\mathcal{S}_{\mathbb{D}}, \mathcal{S}_{\mathbb{D}'}$ by producing the stream $s' \in \mathcal{S}_{\mathbb{D}}$ having an event at timestamp t of each event in s_2 with the value of the latest event in s_1 at timestamp $t' < t$. In other words, whenever there is an event e in s_2 the operator produces an event with the latest value of s_1 known before e appeared.

Delay The **delay** operator of the type $\mathcal{S}_{\mathbb{T}\setminus\{0\}} \times \mathcal{S}_{\mathbb{D}} \rightarrow \mathcal{S}_{\mathbb{U}}$ takes two streams $s_1 \in \mathcal{S}_{\mathbb{T}\setminus\{0\}}$ and $s_2 \in \mathcal{S}_{\mathbb{D}}$ as input. s_1 can be seen as the signal setting a timer, and s_2 as the reset signal cancelling a running timer. Whenever s_2 has an event or the operator produced an output event, an event in s_1 will set a timer. When t was the timestamp when the timer was set and t' the value of s_1 at t, then the operator will produce an event at timestamp $t + t'$ iff there is no reset event on s_2 at time r with $t < r < t + t'$.

Examples

Like for CQL and SQL for PipelineDB we will take the examples from appropriate sources to assure they reflect common use cases the language was intended for. As we will see, the use cases of TeSSLa distance themselves from the ones we will find for the other languages, which is why we will take an example of each area for the comparison in the following chapters. Here, for TeSSLa, we will take runtime verification examples from the TeSSLa web IDE [21]. The first example observes some program using “add” and “sub” functions where “add” should not be used more frequently than “sub”. An error should be indicated if the number of “add” calls exceeds the number of “sub” calls by 2 or more:

```
@InstFunctionCall("add")
in add: Events[Unit]

@InstFunctionCall("sub")
in sub: Events[Unit]

def add_count := count(add)
def sub_count := count(sub)
def diff := add_count - sub_count

def error := diff >= 2

out add_count # signal
out sub_count # signal
out diff # signal
out pure(error) as error # signal
```

The second example is about detecting race conditions in an application using multiple threads to read and write on a single memory section. If one of the threads reads, it will

also write afterwards, thus there is a race condition whenever there are two consecutive read accesses without a write in between:

```
@InstFunctionCall("readValue")
in read: Events[Unit]

@InstFunctionCall("writeValue")
in write: Events[Unit]

def timeRead := time(read)
def timeWrite := default(time(write),0)

def error := unitIf(prev(timeRead) > timeWrite)

out write # unit events
out read # unit events
out error # unit events
```

These examples used inbuilt functions like `count`, counting the events of a stream, `pure`, filtering out events with the same value as the previous event, `default`, defining a default value for a stream before there is the first event, `unitIf`, producing a unit stream with events whenever the defined expression takes the value TRUE and `prev`, keeping the second last event of a stream. They are built in functions which can be used with the TeSSLa interpreter, but they could as well be defined using basic operators. Further expressions like `add_count − sub_count` or `diff >= 2` use the concept lift on a *signal*: The events of a stream are interpreted as changes to the signal's value which is lifted for example by the binary function ">=" to a boolean value. When we interpret a stream as a signal or just a plain event stream in further examples, we might represent them as in Figure 2.1 showing example input and output streams for the add/sub example. There

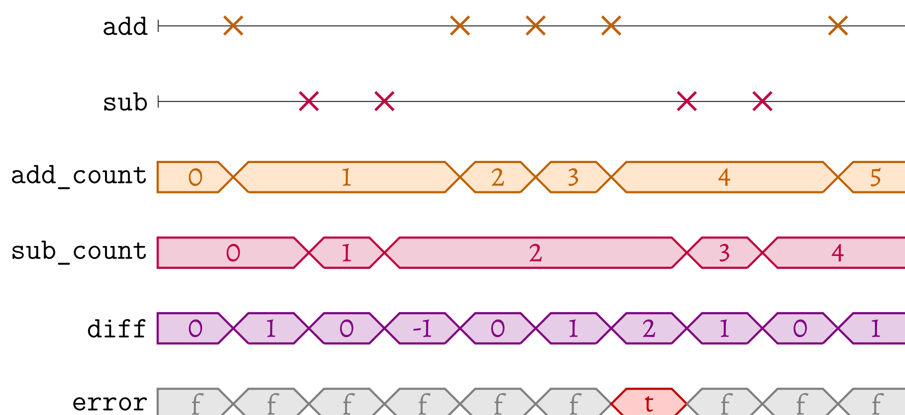


Figure 2.1: Streams for add/sub Example showing Representations of Signal and Events Interpretation of Streams

are many more language features supported by TeSSLa's interpretor and compiler which we cannot cover here but which we might use for the example implementations later. See the original TeSSLa documentation [20] for more information.

2.2 CQL

The Continuous query language [1] takes a relational query language and adds the concept of streams. Both, streams and relations, feature a schema as well as they are dependent on a time domain $\mathbb{T}$.

Basic Concepts and Operators

Before we start to introduce the basic operators, we need to outline our terminology a bit further:

Schema When a tuple s follows a certain schema $\mathbb{S}$, i.e. $s \in \mathbb{S}$, the schema tells for each element of the tuple, to which domain it belongs. More precisely a schema can be described as $\mathbb{S} = \{s = (a_1, a_2, ..., a_n) | a_1 \in \mathbb{D}_1, a_2 \in \mathbb{D}_2, ...a_n \in \mathbb{D}_n\}$ where a_i is a named attribute with domain $\mathbb{D}_i$.

Stream A stream S is a collection of events where each event consists of a value s and a timestamp τ. A stream in the context of CQL cannot only contain multiple events with the same value, moreover multiple events can feature the same timestamp. Thus, with a certain schema $\mathbb{S}$ and a time domain $\mathbb{T}$, a stream S is a multiset of tuples $\langle s, \tau \rangle$ where $s \in \mathbb{S}$ and $\tau \in \mathbb{T}$.

Relation The simplest view on a relation is probably a table where each row is an entry and each column has a certain type, i.e. the columns define a schema. Since these tables are not fixed but can be updated by adding, removing or changing rows, a relation in a system might contain different entries at different points in time. Therefore we define a relation R as a mapping from a time domain $\mathbb{T}$ to a multiset of elements from a schema $\mathbb{S}$.

Since CQL operates on streams and relations, it defines operators to produce one of these types out of the other. Figure 2.2 shows an overview of the available operator classes. CQL does not introduce operators solely working on one of these types since operators taking

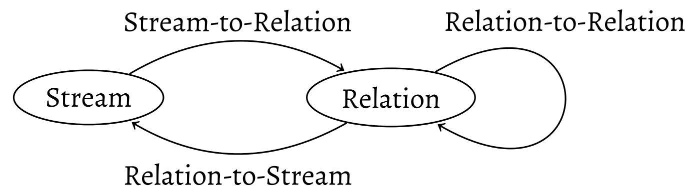


Figure 2.2: CQL Operator Classes

a relation and yielding a relation are already defined in the underlying relational language; operators taking a stream and directly producing a stream are not needed since they can be composed by first turning a stream into a relation, changing this relation and finally producing a new stream out of the obtained relation. We will see examples of this later.

Stream-to-Relation A Stream-to-Relation operator takes a stream as defined above, applies some sort of window and takes a snapshots at different $\tau \in \mathbb{T}$ as the output relation $R(\tau)$.

- Time-based sliding windows The idea is to define the window by some time interval T which, at a certain time τ, includes all events that have appeared not more than T time units in the past. So given a stream S and a time interval T, it produces $R(\tau), \tau \in \mathcal{T}$ with $s \in R(\tau)$ iff $\langle s, \tau' \rangle \in S$ and $\tau - T \leq \tau' \leq \tau$. With T being the *Range* of the window, it can be extended by a second time interval parameter, *Slide*, defining the step size in which the window slides.
- Tuple-based sliding windows As the name implies, rather than looking in the past for a fixed interval in the time domain, it looks in the past for a certain number of events: Given a stream S and a positive integer N, the operator produces a relation R where $R(\tau)$ consists of the N tuples from S which appear in S with the latest timestamps up to τ. Note that since there might be multiple events with the same timestamp, the output might be ambiguous and might further, determined by the implementation, depend on the order of the incoming events.
- Partitioned windows Again, we use the tuple-based approach but now we additionally take a look at certain values of the events: Given S, a positive integer N and a list of attributes from the schema, it produces a relation one would receive by first splitting the stream S into a sub stream for every unique value for each of the given attributes then separately applying a tuple-based sliding window and finally performing a union over all resulting windows. Again, just as in the plain tuple-based sliding window, the output might be ambiguous.

Relation-to-Stream The Relation-to-Stream operators produce their output based on when the tuples are in the relation and when they are not, or in other words, when they are inserted or removed. The following descriptions will assume the time domain to be integer.

- Istream ("insert stream") A tuple will appear in the output stream S at the first time it appeared in the input relation R: $\langle s, \tau \rangle \in S$ iff $s \in R(\tau)$ and $s \notin R(\tau - 1)$.
- Dstream ("delete stream") A tuple will appear in the output stream S when it was removed, meaning at the first time it is *not* part of the input relation R: $\langle s, \tau \rangle \in S$ iff $s \in R(\tau - 1)$ and $s \notin R(\tau)$.
- Rstream ("relation stream") A tuple will appear in the output stream S whenever it is in the input relation R: $\langle s, \tau \rangle \in S$ iff $s \in R(\tau)$.

Examples

In our examples we will assume the underlying relational language to be SQL. The following use cases and queries have been taken from the Stream Query Repository [19]. The

repository is subdivided into different application domains including Road Traffic Monitoring with the Linear Road Benchmark which we will look into in later chapters; here we will just take small examples out of two other application domains to give a short expression: One of the queries presented for *online auctions* is the “Hot Item Query” asking for the most popular items determined by the number of bids over the last hour, which should be updated every minute:

```
HotItemStream:
    Select Rstream(itemID)
    From   (Select   B1.itemID as itemID, Count(*) as num
            From     Bid [Range 60 Minute
                          Slide 1 Minute] B1
            Group By B1.itemID)
    Where   num >= All (Select   Count(*)
                        From     Bid [Range 60 Minute
                                      Slide 1 Minute] B2
                        Group By B2.itemID)

Select *
From   HotItemStream [Range 1 Minute]
```

For *Network Traffic Management* one of the queries they suggest is the “Protocol Analysis Query” summing up the total length and number of packages of HTTP requests, which are assumed to use port 80, per source IP in non-overlapping 5 minute intervals:

```
Select   Rstream(srcIP, Sum(len), Count(*))
From     Packets [Range 5 Minute
                  Slide 5 Minute]
Where    destPort = '80'
Group By srcIP
```

2.3 SQL for PipelineDB

PipelineDB [13] is an open source extension for PostgreSQL [16] designed to run SQL queries continuously on data streams. As discussed in [10], plain SQL is not complete for streaming applications, an issue which can be solved using UDAs, and PipelineDB is an extension to PostgreSQL which itself features UDAs. On top of that, PipelineDB introduces some special continuous aggregates as well as a time-based window function. Since basic SQL is expected to be common, the following explanation will only present a short impression of the original PipelineDB documentation [14]; furthermore code examples within the description are also taken from there.

Basic Concepts

The model of data streams used for PipelineDB might be seen as append only bags of tuples, i.e. an append only table, which is exactly what one would retrieve by setting up a

view selecting every single event from a stream as it is. If all defined structures listening to a stream have processed a stream event, the event itself is discarded. This allows the system to only store data needed by the views and operators and save resources.

Streams To access a data stream, it needs to be defined first:

```
CREATE FOREIGN TABLE stream_name ( [
   { column_name data_type [ COLLATE collation ] } [, ... ]
] )
SERVER pipelinedb;
```

Similarly to a usual table, this defines the stream "stream_name" with its schema on the server "pipelinedb". Feeding events to a stream can be done the same way as filling a table: simply use `INSERT` statements, `COPY` from a source or use any client compatible with PostgreSQL.

Continuous Views Continuous views can be specified as usual views with `action` set to `materialized` which, since it is default, can be omitted.

```
CREATE VIEW name [WITH (action=materialize [, ...])]  AS query
```

From the documentation, query is a SELECT statement limited to

```
 SELECT [ DISTINCT [ ON ( expression [, ...] ) ] ]
    expression [ [ AS ] output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]

-- where from_item can be one of:
    stream_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    table_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item [ NATURAL ] join_type from_item [ ON join_condition ]
```

Windows Apart from using Time-to-Live Expiration on views for their entries, one can also use the built-in time-based sliding window like in the following example:

```
 CREATE VIEW recent_users WITH (sw = '1␣minute') AS
   SELECT user_id::integer FROM stream;
```

Which internally will be rewritten to

```
 CREATE VIEW recent_users AS
   SELECT user_id::integer FROM stream
WHERE (arrival_timestamp > clock_timestamp() - interval '1␣minute');
```

The documentation does not mention any tuple-based sliding windows known from CQL; Although looking at the last example might bring up the idea to create one using a descending `ORDER` on the "arrival_timestamp" and a `LIMIT` of *N*, they cannot be implemented this easily, since, as the above description suggests, continuous views do not support these operators.

Continuous Transforms Continuous transforms can be used for stream-to-stream transforms without necessarily storing the events. They are defined similar to continuous views:

```
CREATE VIEW name (WITH action=transform [, outputfunc=function_name(
    arguments ) ]) AS query
```

Again, query is a SELECT statement with the same limitations as for continuous views. Additionally, aggregates cannot be used in any of the used expressions. "outputfunc" is a so called trigger function, applied to every single event outputted by the transform. Currently, PipelineDB only provides one of these as a built-in, namely `pipelinedb.insert_into_stream` which takes a stream label and does exactly what the name suggests. Custom trigger functions can also be defined, the following example writes the output to a table instead of another stream:

```
CREATE TABLE t (user text, value int);

CREATE OR REPLACE FUNCTION insert_into_t()
  RETURNS trigger AS
  $$
  BEGIN
    INSERT INTO t (user, value) VALUES (NEW.user, NEW.value);
    RETURN NEW;
  END;
  $$
  LANGUAGE plpgsql;

CREATE VIEW ct WITH (action=transform, outputfunc=insert_into_t) AS
  SELECT user::text, value::int FROM stream WHERE value > 100;
```

Additionally PipelineDB supports continuous joins and numerous built in aggregates for continuous views such as almost all of those defined in standard PostgreSQL, whereas some of them behave a little different to operate non-blockingly, as well as some extra aggregates specifically designed for streaming data like for bloom filters, frequency tracking or top-k functionality, just to name a few.

Examples

The following queries are taken from [15]. As common use cases they name *realtime reporting dashboards* with examples like

```
-- Calculate the number of unique users seen per url referrer each day
   using only a constant amount of space per day
CREATE VIEW uniques AS
    SELECT
        day(arrival_timestamp),
        referrer::text,
        COUNT(DISTINCT user_id::integer)
    FROM users_stream GROUP BY day, referrer;

-- How many ad impressions have we served in the last five minutes?
CREATE VIEW imps WITH (sw = '5␣minutes') AS
    SELECT COUNT(*) FROM imps_stream
```

and *realtime monitoring systems* to react on certain events such as if the server latency is too high or some users generate too much traffic:

```
-- What are the 90th, 95th, and 99th percentiles of my server's request
   latency?
CREATE VIEW latency AS
    SELECT
        percentile_cont(array[90, 95, 99])
        WITHIN GROUP (ORDER BY latency::integer)
    FROM latency_stream;

-- Heavy hitters: how much traffic are each of the top-10 IP addresses
   making requests to my server generating?
CREATE VIEW heavy_hitters AS
    SELECT
        day(arrival_timestamp),
        topk_agg(ip, 10, response_size)
    FROM requests_stream GROUP BY day;
```

`percentile_cont` and `topk_agg` are two of PipelinDB's inbuilt continuous aggregates.

3

Example Use Cases

To compare the languages, we will use two use cases: One of them is the Linear Road Benchmark where we will find queries involving saved data which can be implemented naturally in SBDs and on the other hand we will introduce a burst pattern example which has a focus on timing.

3.1 Linear Road Benchmark

This section is based on the original paper [2]. They designed the benchmark to compare performance characteristics of DSMSs in running continuous queries as well as queries addressing historical data. Since, as described earlier, there is no standard in streaming applications, there were several challenges to face. First of all, to ensure that the benchmark examines practically relevant demands, they should provide *semantically valid input* for the systems and not just random values. Therefore they picked the use case of a variable tolling system which should calculate the tolls to be charged for each vehicle based on position reports generated using the MIcroscopic Traffic SIMulator (MITSIM) [24]. In the following subsection we will describe the setting in more detail.

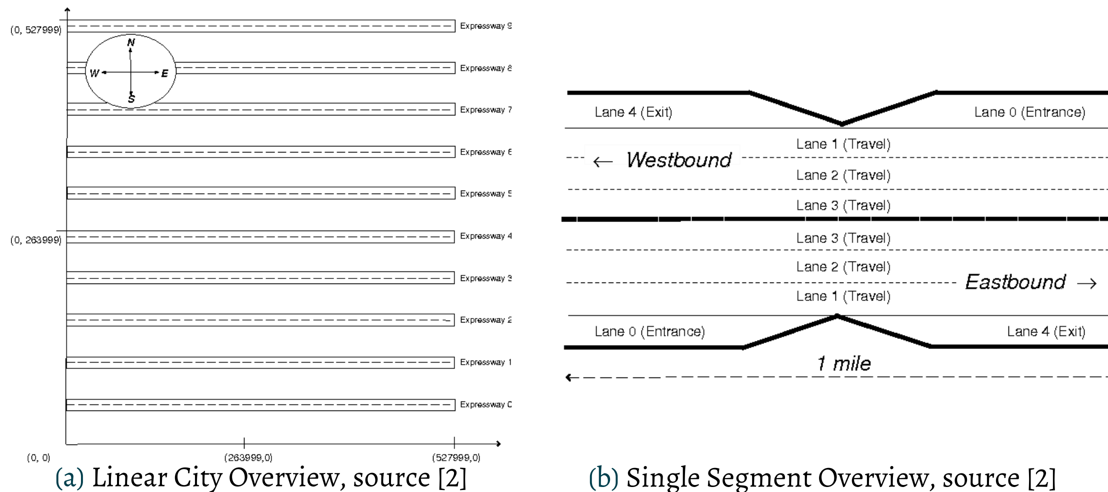
Now that the setting and input is set, the question arises, how to specify the desired output of the system in the absence of an appropriate *standardized language*. The semantics should get clear, however choosing an existing query language like CQL implies the risk of influencing the specification through features of that language. Instead they naturally outlined the requirements with supportive descriptions in predicate calculus.

Moreover, the queries they defined have *ambiguous correct answers* in general, which might depend on the arrival order of events and the implementation. To still be able to check the system's answer for correctness, they built a validator tool.

Another open question was, how to define *continuous query performance metrics*. One could take the response time into account, i.e. how long it will take the system to produce output from a trigger event in average or at maximum. Another metric is the supported query load which asks how much input the system can handle while still returning correct results and meeting response time constraints. They decided to introduce the L-Rating which reflects the supported query load by indicating the maximum number of expressways the system could handle without violating response constraints.

Linear Road Setting

Linear City has a number of expressways which, for simplicity, all run between east and west. Each of them consists of three travel lanes per direction. Their full length of 100 miles is subdivided in one-mile segments where each of these segments feature an entrance and exit lane for both directions. Figure 3.1 (a) shows an overview of the city's expressways and Figure 3.1 (b) illustrates a single segment. The idea is, that a toll should



(a) Linear City Overview, source [2]

(b) Single Segment Overview, source [2]

Figure 3.1: Linear City's Expressways.

be charged for every vehicle on the expressways where the amount depends on factors such as the number of vehicles or average velocity within the corresponding segment and whether there is an accident nearby. A car should be notified whenever there is a toll to be charged if it proceeds to the next segment and it should be warned if it approaches an accident area. Additionally every vehicle might issue information requests for already charged tolls or travel estimations at any time.

System Input

The input is partitioned in *historical data*, which might be stored in the system before the actual benchmark starts, and *stream data* used for continuous queries as well as for query requests accessing historical data. All entries, historical and stream data, are composed of positive integer values, and -1 denotes NULL.

Historical Data consists of two parts:

- The *Toll History*'s entries feature the following schema:

  (VID, Day, XWay, Tolls)

  For each vehicle in the simulation, addressed by its vehicle ID (VID), and every expressway (XWay), it contains an entry for the charged tolls for each day in the past ten weeks. Since MITSIM generates roughly 150000 vehicles in a simulation, the toll history comprises about $150000 \cdot 10 \cdot 7 \approx 10$ million entries for one expressway.

– The *Segment History*'s entries can be described as follows:

(Day, Min, XWay, Dir, Seg, Lav, Cnt, Toll)

Analogous to the toll history, this covers the past ten weeks and for each Minute it contains an entry for every Segment for each Direction on every expressway, giving information of the number of vehicles (Cnt), the average velocity (Lav) and the Toll to charge. Since an expressways consists of two directions, each separated in 100 segments, the segment history contains about $200 \cdot 10 \cdot 7 \cdot 24 \cdot 60$ entries, which is about 20 million – again, just for a single expressway.

Stream Data also features different types of events, namely *Position Reports*, *Account Balance Request*, *Daily Expenditure Request* and *Travel Time Request*, which all follow the same schema. Table 3.2 sums up, which of the fields are used for the different event types or which are unused and in the simulation therefore might be labeled with −1. To distinguish between the different types and to know which values of the event

	Type	Time	VID	Spd	XWay	Lane	Dir	Seg	Pos	QID	S_{init}	S_{end}	DOW	TOD	Day
Position Report	0	*t*	*v*	*spd*	*x*	*l*	*d*	*s*	*p*	-1	-1	-1	-1	-1	-1
Acc. Bal. Req.	2	*t*	*v*	-1	-1	-1	-1	-1	-1	*q*	-1	-1	-1	-1	-1
Daily Exp. Req.	3	*t*	*v*	-1	*x*	-1	-1	-1	-1	*q*	-1	-1	-1	-1	*n*
Trav. Time Req.	4	*t*	*v*	-1	*x*	-1	-1	-1	-1	*q*	*org*	*dst*	*wd*	*dt*	-1

Table 3.2: Input Stream Event Types of Linear Road

should actually be kept, all events start with a number indicating their type. The next two values always tell from which vehicle the event was issued, addressing it by its unique Vehicle ID, and the time of the event in seconds from the start of the simulation. Position reports will be sent from every vehicle on the expressways exactly every 30 seconds. Since all expressways run between east and west, the position in north-south direction is given by the expressway, lane and the side of the road, indicated by the direction in which the vehicle is traveling. The east-west direction is covered by the exact position in feet (see Figure 3.1(a)) as well as redundantly by the current segment, since it is used in many of the benchmark's computations. On top of that, a position report features the speed at which the vehicle is moving. These values are assumed to be valid for the whole time from the previous 30 seconds, including the second *t* of the report. The other events are query requests featuring a unique Query ID. How they should be handled will be outlined in the following subsection.

The stream data for a simulation covers a three hour period and contains about 12 million events of which only a small part is query requests: Every time a position report is generated, with a chance of 1% there will also be generated a query request which will be an account balance request with a chance of 0.5, a daily expenditure

request with 0.1 and with a probability of 0.4 it will be a travel time estimation request. Accidents, which are not explicitly indicated by the simulation input, can be observed with the position reports. For each expressway in the simulation MITSIM generates an accident every 20 minutes at a random position, which will be cleared after 10 to 20 minutes.

System Output

Apart from query answers which are triggered by the corresponding request, the system should also send notifications triggered by position reports if certain preconditions are fulfilled. As mentioned earlier, in the original paper they introduced notations based on predicate calculus to provide a precise specification which we will borrow here (see Table 3.3) to keep this section detailed, yet brief. Let P denote the set of all position reports and

$$
\begin{aligned}
cars(m,x,s,d) &= \{p.\text{VID} \mid p \in P,\ m = M(p.\text{Time}),\\
&\quad\ p.(\text{XWay}; \text{Seg}; \text{Dir}) = (x; s; d)\}\\
Avgsv(v,m,x,s,d) &= \text{AVG}(\{|p.\text{Spd} \mid p \in P,\ p.\text{VID} = v,\ m = M(p.\text{Time}),\\
&\quad\ p.(\text{XWay}; \text{Seg}; \text{Dir}) = (x; s; d)|\})\\
Avgs(m,x,s,d) &= \text{AVG}(\{|Avgsv(v,m,x,s,d) \mid v \in cars(m,x,s,d)|\})\\
Lav(m,x,s,d) &= \lfloor \text{AVG}(\{|Avgs(m-1,x,s,d), \ldots, Avgs(m-5,x,s,d)|\}) \rfloor\\
Toll(m,x,s,d) &= \begin{cases} 2 \cdot (|cars(m,x,s,d)| - 50)^2 \\ \quad \text{if} \quad Lav(m,x,s,d) < 40 \text{ and} \\ \qquad |cars(m,x,s,d)| > 50 \text{ and} \\ \qquad \forall_{0 \le i \le 4}(\neg(Acc_in_Seg(m-1,x,Dn(s,d,i)))) \\ 0, \quad \text{otherwise} \end{cases}\\
Stop(v,t,x,l,p,d) &\Leftrightarrow \forall_{1 \le i \le 4}(Last_i(v,t).(\text{XWay}; \text{Lane}; \text{Pos}; \text{Dir}) = (x,l,p,d))\\
Acc(t,x,p,d) &\Leftrightarrow \exists_{v_1,v_2,l}(l = \text{TRAVEL} \wedge v_1 \neq v_2 \wedge Stop(v_1,t,x,l,p,d) \wedge\\
&\quad\ Stop(v_1,t,x,l,p,d))\\
Acc_in_Seg(m,x,s,d) &\Leftrightarrow \exists_{p,t}\left(t \in m \wedge Acc(t,x,p,d) \wedge \lfloor \frac{p}{5280} \rfloor = s\right)
\end{aligned}
$$

Table 3.3: Notations for Linear Road's Toll and Accident Definitions

p be a tuple, for example a position report, i.e. $p \in P$. Along $p.\text{Time} = t$ they use the the shorthand notation

$$p.(\text{XWay}; \text{Seg}; \text{Dir}) = (x; s; d) \Leftrightarrow p.\text{XWay} = x \wedge p.\text{Seg} = s \wedge p.\text{Dir} = d.$$

For each $p \in P$ they define $\overleftarrow{p}$ as the last position report emitted by the same vehicle within the same trip before p:

$$\overleftarrow{p} = q \in P \text{ s.t. } (q.VID = p.VID \wedge p.Time - q.Time = 30)$$

and similarly $\vec{p} = q \in P$ s.t. $\overleftarrow{q} = p$. Further

$$M(t) = \lfloor \frac{t}{60} \rfloor + 1$$

is the number of the minute containing t, for which they also use $t \in m$ if $M(t) = m$. For a non-negative integer i,

$$Dn(s,d,i) = \begin{cases} \text{MIN}(s+i, 99) & \text{if } d = 0 \\ \text{MAX}(s-i, 0) & \text{otherwise} \end{cases}$$

is the i-th segment after segment s in direction $d \in \{0 = \text{eastbound}, 1 = \text{westbound}\}$. For a time t, a positive integer i and a vehicle identified by v,

$$Last_i(v,t) = p \in P \text{s.t.} (p.\text{VID} = v \wedge 30(i-1) \leq t - p.\text{Time} < 30i)$$

denotes the i-th last position report emitted by vehicle v until second t. Finally, $\{|...|\}$ is the content of a set and $l = \text{TRAVEL}$ means that l is a travel lane i.e. $l \in \{1,2,3\}$ as well as $l = \text{EXIT} \Leftrightarrow l = 4$, according to Figure 3.1 (b).

The system's output is divided into the following event types:

Toll Notifications When a vehicle uses a certain segment of an expressway it will cost a certain toll. Since it might be variable, it should be informed about the toll's amount whenever a vehicle enters a new segment so it can leave the expressway before it is actually charged. A trigger for this type of event is therefore a position report q indicating a change of segment $q.\text{Seg} \neq \overleftarrow{q}.\text{Seg}$ and which is not showing the vehicle leaving the expressway: $q.\text{Lane} \neq \text{EXIT}$.

$$(\text{Type: } 0, \text{VID: } v, \text{Time: } t, \text{Emit: } t', \text{Spd: } Lav(M(t),x,s,d), \text{Toll: } Toll(M(t),x,s,d))$$

While some of the answer's values are directly taken from the triggering position report, $(v,t) = q.(\text{QID}; \text{Time})$, the Speed and Toll values are determined using the values $(t,x,s,d) = q.(\text{Time}; \text{XWay}; \text{Seg}; \text{Dir})$. Speed is calculated as the "Latest Average Velocity" (*Lav*), which is the average velocity at segment s over the past five minutes over all cars which have sent a corresponding position report. The amount of the *Toll* is zero if the *Lav* is at least 40, the current number of *cars* at the segment does not exceed 50 or there is an accident ahead within the following four segments or in segment s.

Accident Alerts Accidents are defined to happen when at least two vehicles are stopped on one of the travel lanes at the exact same position. A vehicle in turn is said to be stopped whenever, determined by the (x,l,p,d)-position, it has not moved in the time of its last four reports. Whenever there is an accident, every vehicle approaching the accident area should be informed. Thus the precondition for a position report to trigger an accident alert is first of all the same as for toll notifications, $q.\text{Seg} \neq \overleftarrow{q}.\text{Seg} \wedge q.\text{Lane} \neq \text{EXIT}$, but additionally there must be an accident segment s' ahead: $\exists_{s', 0 \leq i \leq 4}(s' = Dn(q.\text{Seg}, d, i) \ \wedge \ Acc_in_Seg(M(t)-1, x, s', d))$.

$$(\text{Type: } 1, \text{Time: } t, \text{Emit: } t', \text{Seg:} s')$$

In Section 4.2 we will examine accident alerts in greater detail and also present Figure 4.1 of an example accident's timing properties taken from position reports of a simulation created with MITSIM. Note that in general there might be multiple accident segments for which a single position report should trigger alerts for. Since

MITSIM only generates an accident every 20 minutes at a random position over the total number of 200 segments per expressway (100 in each direction), which will be cleared *at the latest* after 20 minutes, this is a very unlikely, if even possible scenario in the simulations. Still we will consider this for our implementation later, because it cannot be solved trivially with TeSSLa.

Account Balances The account balance is defined as the sum of all charged tolls of the current day until now, which is from the start of the simulation. "Until now" in this case means that the value must have been valid for a time τ which is not too far in the past, or more precisely $\tau \geq t - 60$ where $t = r.\text{Time}$ with r being the triggering request event.

$$(\text{Type: } 2, \text{Time: } t, \text{Emit: } t', \text{ResultTime: } \tau, \text{QID: } q, \text{Bal: } \mathit{tollsum}(v, \tau))$$

As pointed out earlier, tolls for a segment are only charged when a vehicles proceeds through the segment's bounds to the next segment, thus the balance value *tollsum* can be defined as follows:

$$\mathit{tollsum}(v,t) = \sum_{\substack{p \in \mathit{tollset}(v) | \\ p.\text{Time} \leq \tau \wedge \\ p.\text{Seg} \neq \mathit{Last}_1(v,t).\text{Seg}}} \mathit{Toll}(p)$$

with $\mathit{Toll}(p) = \mathit{Toll}(M(p.\text{Time}), p.\text{XWay}, p.\text{Seg}, p.\text{Dir}))$ and $\mathit{tollset}(v)$ being the set of all position reports of v where the vehicle left a segment without leaving the expressway:

$$\mathit{tollset}(v) = \{p \in P | p.\text{VID} = v, p.\text{Seg} \neq \vec{p}.\text{Seg}, p.(\text{XWay}; \text{Dir}) = (x; d)\}$$

Daily Expenditures The daily expenditure is defined similar to the account balances but instead of asking for the total sum of the current day, it is the sum of all tolls on a certain expressway on a day in the past ten weeks:

$$(\text{Type: } 3, \text{Time: } t, \text{Emit: } t', \text{QID: } q, \text{Bal: } \mathit{tollsum}(\text{v,n,x}))$$

with

$$\mathit{tollsum}(v,n,x) = \sum_{\substack{p \in \mathit{tollset}(v) | \\ \mathit{Day}(p.\text{Time}) = n \wedge \\ p.\text{XWay} = x}} \mathit{Toll}(p).$$

Travel Time Estimations Given the statistics over the previous ten weeks, a travel time estimation request is to be answered by

$$(\text{Type: } 4, \text{QID: } q, \text{TravelTime: } r_1, \text{Toll: } r_2)$$

where r_1 and r_2 are the estimated time and tolls it will cost to travel from the initial segment *org* to the end segment *dst* starting at time *dt* of a day of week *wd*. With

$$y_{org} = dt$$

and

$$y_{i+1} = y_i + tav(x, i, n, y_i) \qquad \forall org < i \leq dst$$

where $tav(x, i, d, y)$ estimates the travel time through the segment i on expressway x based on the historical average velocities at time y on a day of week d, the estimated travel time as the arrival time at destination segment is given by $r_1 = y_{dst}$ and the expected total toll can be defined as

$$r_2 = \sum_{i=org}^{dst-1} cav(x, j, d, y_i)$$

where $cav(x, j, d, y_i)$ estimates the toll based on the historical data according to *Toll* in Table 3.3.

The response time of the system is the difference between the time of the triggering event (Time: t) and the time the answer is emitted (Emit: t'). Table 3.4 shows the maximum response time constraints for the different output events defined for the benchmark.

	Toll Notif.	Accident Al.	Account Bal.	Daily Exp.	Travel Time Est.
$t' - t \leq$	5 Seconds	5 Seconds	5 Seconds	10 Seconds	30 Seconds

Table 3.4: Response Time Constraints for Linear Road's output Event Types

3.2 Burst Pattern

Since the Linear Road Benchmark was designed for general DSMSs, our second use case will target timing patterns for which TeSSLa was specifically designed. As TeSSLa allows to write libraries and they included a standard library into the interpreter which covers different timing patterns including burst patterns, with TeSSLa we will be able to solve this example using inbuilt functions. However, to aim for a fair comparison of the specifications in the different languages, we will also break the used TeSSLa inbuilts down to basic operators. Since the burst pattern is only a *pattern* and therefore less concrete as our linear road *use case*, we will outline this pattern based on the AUTOSAR Timing Extension according to a COEMS timing tutorial [7] and concretize it as a simple example from TeSSLa web IDE [21].

Timing Pattern

For the Burst Pattern a certain event should, as the name suggests, only appear in bursts: The first event will start a new burst frame which has the size defined by *burst length*, 3 seconds for example. Whenever there is another event in the frame, i.e. within the next 3 seconds, it belongs to the same burst. When a new event appears outside a burst frame it starts a new burst. This simple pattern does apply as long as the number of events of each burst does not exceed the *burst amount* e.g. 4, see Figure 3.5, and it is said to be violated if

Figure 3.5: Simple Burst Pattern, example from [7]

the burst amount is exceeded. We extend this pattern by a *waiting period* e.g. 2 seconds directly following a burst frame in which there must be no events. A stream following this pattern is illustrated in Figure 3.6. Finally, for our example, we introduce a *silencing*

Figure 3.6: Burst Pattern with Waiting Period, example from [7]

condition which starts and ends checking the stream for the burst pattern by taking the value FALSE and TRUE respectively. It is called silencing condition, since if its value is TRUE the input stream should be silenced, i.e. instead of checking if it fulfills the burst pattern, we check whether there is not a single event. Figure 3.7 shows this property.

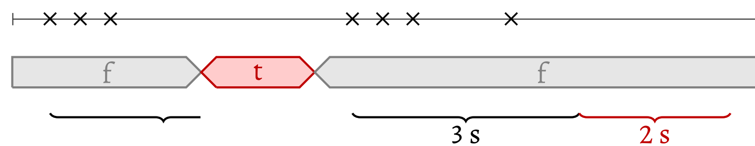


Figure 3.7: Burst Pattern with Waiting Period and Silencing Condition. As implied by the Braces, Bursts and Waiting Periods are cancelled by a Change in the Silencing Condition's Value.

Burst Pattern Example

As mentioned above, we will describe the TeSSLa Example "Burst Pattern" from [21].

Input As Input we have two streams, a and b respectively, with events of integer values and a stream we will evaluate the burst pattern on: e. Since only the time of the events is relevant for checking on the timing property burst pattern, e's events could have any value. Therefore the domain chosen for e events in TeSSLa is $\mathbb{U}$, the unit domain.

Evaluation The task is to compute the timing property p as the burst pattern with *burst length* of 2 seconds, *waiting period* of 1 second, *burst amount* of 3 and a *silencing condition* which is defined using a and b: Whenever the last known event of a has a value greater than the the last known event's value from stream b, the silencing condition is TRUE, i.e. there should be no events on stream e. In the case that the last known value of a is less than or equal the last known of b the silencing condition is FALSE, then we will check for the burst pattern on e. From the beginning as long as there

has not yet been an event on each of the streams a and b, the value of the silencing condition is UNKNOWN and events on e are ignored. The value of p shall be UNKNOWN if the value of the the silencing condition is UNKNOWN and TRUE if the burst pattern with known value of the silencing condition is met. If the silencing condition's value is known and the burst pattern described above is violated, i.e. there was an event on e while it should be silenced, a single burst features too many events or there is an event in the waiting period after a burst, then the value of the property p is FALSE.

Output All input events of the streams a, b and e should be outputted as well as the value of p whenever it changes.

The example input and output streams given at [21] is illustrated in Figure 3.8 where c is the silencing condition.

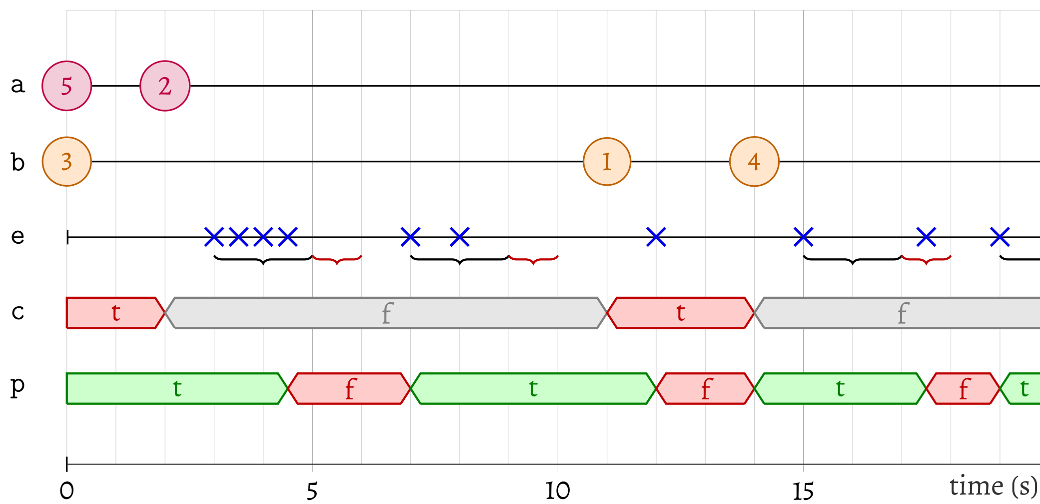


Figure 3.8: Burst Pattern on Example Trace from [21]

4

Stream Property Implementations

We will start implementing the examples from Chapter 3 with Linear Road's Daily Expenditure requirements as it shows how the languages might be used to access historical data. Next we will proceed to the Accident Alerts since they combine accessing potentially unlimited datasets with simple ordering and timing patterns. Finally we will implement the Burst Pattern example as a classical timing pattern for TeSSLa. The implementations will be quite detailed and, to some extend, consider edge cases. Therefore we will summarize them in the end of the chapter.

4.1 Linear Road: Daily Expenditures

Daily Expenditure requests address a set of historical data and, as we will see, this can be implemented naturally with our SDB query languages. The historical data should cover the past 10 weeks, thus the set should theoretically be updated, when the time of the simulation exceeds one day, but this will not happen by running the benchmark since its simulations start on second 0 of a new day and only cover 3 hours.

CQL

As mentioned earlier, the Stream Query Repository [19] contains a section about the Linear Road Benchmark with query implementations in CQL, however by the time they were written, the specifications were not yet complete, so the presented queries are also incomplete. For this and the following queries we will therefore take the suggested queries and change or extend them if needed. First of all, we will start by defining input streams and adapt them to the schemas they used as far as possible. As the stream events are given in one schema using "NULL" values for unused fields, we will take the input stream of events issued by the cars and extract the substream of Daily Expenditure Requests:

```
CarEventStr(type,   /* 0,2,3,4                        */
            time,   /* 0..10799 for 3h simulations */
            carid,  /* unique Vehicle IDentifier   */
            speed,  /* speed of the car            */
            xway,   /* expressway: 0..L-1          */
```

```
            lane,    /* lane: 0..4                     */
            dir,     /* direction: 0(east), 1(west) */
            seg,     /* segment: 0...99                */
            pos,     /* coordinate in express way:
                      * 0..527999                      */
            qid,     /* id used to associate
                      * responses with queries         */
            m_init,  /* initial segment                */
            m_end,   /* final segment                  */
            dow,     /* day of week                    */
            tod,     /* time of day                    */
            day);    /* day from yesterday to 10
                      * weeks ago: 1..69               */
ExpQueryStr:
    SELECT RSTREAM(time,carid,xway,qid,day)
    FROM CarEventStr [NOW]
    WHERE type = 3;
```

Next we define the historical data tables to address with the queries.

```
TollHistStr(carid, day, xway, tolls);
TollHistory:
    SELECT *
    FROM TollHistStr [UNBOUNDED];
```

Note that we do not consider updating the toll history which would require us for the `TollHistStr` to unite the toll history input with new historical toll entries whenever another day ends. Then, to omit old data and only store relevant entries we would replace the UNBOUNDED window with for example RANGE 10 WEEKS SLIDE 1 DAY. On top of that, we would need to update each entry's day value when today becomes yesterday, yesterday becomes the day before yesterday and so on – which would be too much unnecessary work. Instead, one might think of just counting the days and using the number of the present day as offset to calculate the day which is "day" days in the past. The fact that applying ongoing day numbers for new history entries would require us to go below zero, does not make this approach seem to be intended. And since running the benchmark will never cover the end of the day, we will not implement the history updates, instead we will just leave a note on how it could be done like we did above. The actual output stream for Daily Expenditure Answers can now be specified as follows:

```
ExpOutStr:
    SELECT RSTREAM(3 as type, Q.time, CURRENT_TIME as emit, Q.qid, H.tolls)
        FROM TollHistory as H, ExpQueryStr [NOW] as Q
        WHERE H.carid = Q.carid AND H.day = Q.day AND H.xway = Q.xway;
```

While this query is easy to understand and might feel natural to write, it is questionable what will happen, when the system cannot keep up with too many input events: Since with the [NOW] we are joining the toll history with current timestep's queries, like with

all queries of the current second, the emit time will always equal the query time or, if the system is busy with queries like this for following seconds, it might skip queries of another second completely. As an attempt to face this issue, by considering the response time constraint, one could replace the NOW by RANGE 5 SECONDS and, since this way the requests will stay in the window for longer but should still be answered just once, the RSTREAM must be replaced by an ISTREAM.

SQL for PipelineDB

With PipelineDB there are different ways to implement the specification. For this part we will use continuous views as they make the Daily Expenditures as handy as with CQL. Again we start with defining the car event input stream and selecting required values for the Daily Expenditure Request substream:

```
CREATE FOREIGN TABLE car_event_stream (
    type   INT, time   INT, carid  INT, speed  INT, xway   INT,
    lane   INT, dir    INT, seg    INT, pos    INT, qid    INT,
    m_init INT, m_end  INT, dow    INT, tod    INT, day    INT
)
SERVER pipelinedb;

CREATE VIEW dexp_req WITH (action=transform) AS
    SELECT time, carid, xway, qid, day
    FROM car_event_stream
    WHERE type = 3;
```

For the historical tolls stream and tables, in PipelineDB we could just insert new history entries into the same stream we use for loading the history before starting the simulation, but we would still need to take the same steps as for CQL to omit old data and correctly address days by their numbers.

```
CREATE FOREIGN TABLE hist_tolls_stream (
    carid INT, day   INT, xway  INT, tolls INT
)
SERVER pipelinedb;

CREATE VIEW toll_history WITH (action=materialize) AS
        SELECT carid, day, xway, tolls
 FROM hist_tolls_stream;
```

Now, to obtain the desired query answers, we can simply join the stream of requests with a table, or in this case with the continuous view:

```
CREATE VIEW dexp_answers WITH (action=transform) AS
    SELECT 3 as type, q.time, CURRENT_TIME as emit, q.qid, h.tolls
    FROM output_of('dexp_req') q, toll_history h
        WHERE h.carid = q.carid AND h.day = q.day AND h.xway = q.xway;
```

The direct join between the stream and the table within the continuous transform can handle the stream's events one after another and does not require an explicit window as in CQL, so even if the system is busy processing all input events, there is no time window dependent on the current time which might drop events appearing with too old timestamp; with this implementation, the system will process all requests one after another regardless of the response time. Also, instead of using a continuous transform to output the answers as a stream, we could change it into a continuous view to collect them in a table, by changing the `action` from `transform` to `materialize`.

TeSSLa

For implementing this with TeSSLa, we will first have to take a closer look on time: Since TeSSLa requires the events within its streams to have unique timestamps, but the simulation produces many events for each timestep i.e. every second, we need to introduce a new time domain of finer granularity, for example the time an event is passed to the implementation in nanoseconds from starting the simulation. Further, while timing between events is a key property for TeSSLa, it does not provide a global "current time" function independent of the event's timestamps; thus, as described in Chapter 6, the emit time value will be added to answers in a post-processing step. The input stream's events in TeSSLa will be defined as tuples of integers. To extract the values needed for a substream, the stream is first filtered to only contain Expenditure Requests and then lifted to a stream with another type:

```
in carEventStr: Events[CarEvent]

def dExpReqStr: Events[DExpReq] =
    liftToDExpReq(filter(carEventStr, isDExpReq(carEventStr)))
```

To store the history for later use in answering queries, we need to make use of TeSSLa's unbounded data structures, since there is theoretically no upper bound on the number of cars in a simulation and therefore no bound on entries to store. Essentially we define a stream of maps containing the historical data. Whenever there is a new historical entry, we update the last state of the history known before the new entry appeared, which should be initialized with an empty map.

```
in tollHistStr: Events[TollHistEntry]

def tollHistory: Events[TollHist]=
    slift(
        tollHistStr,
        default(last(tollHistory,tollHistStr),
            Map.empty[Int,Map[(Int,Int),Int]]),
        updTollHistory)
```

The toll history map is structured as follows: An outer map uses the day as a key to get the day's history map. Within this map, the tuple (`carid,xway`) can be used to obtain the corresponding tolls.

```
def dailyExpAnswer=
    slift(
        last(tollHistory,dExpReqStr), dExpReqStr,
        writeDExpAnswer)
```

This implementation of nested maps also allows to easily replace all entries for a single day by just adding a new map of daily expenditures to the outdated day key. These specifications obviously use non built-in data types and functions, which also need to be defined. The presented implementation of `writeDExpAnswer` assumes all toll history entries to be present.

```
type CarEvent={
    type:   Int, time:  Int, carid: Int, speed: Int, xway: Int,
    lane:   Int, dir:   Int, seg:   Int, pos:   Int, qid:  Int,
    m_init: Int, m_end: Int, dow:   Int, tod:   Int, day:  Int}

type DExpReq={
    time:   Int, carid: Int, xway:  Int, qid:   Int, day:  Int}

type TollHistEntry={
    carid:  Int, day:   Int, xway:  Int, tolls: Int}

# Map(day -> Map((carid,xway) -> tolls))
type TollHist = Map[Int,Map[(Int,Int),Int]]

def isDExpReq(stream: Events[CarEvent]): Events[Bool] =
    slift1(stream, (e: CarEvent) => e.type == 3)

def liftToDExpReq(stream: Events[CarEvent]): Events[DExpReq] =
    slift1(stream,
        (e: CarEvent) => {
            time  = e.time, carid = e.carid, xway  = e.xway,
            qid   = e.qid,  day   = e.day})

def updTollHistory(d: TollHistEntry,tH: TollHist) =
    if Map.contains(tH,d.day) then
        Map.add(
            tH, d.day,
            Map.add(Map.get(tH,d.day),
                (d.carid,d.xway), d.tolls))
    else
```

```
        Map.add(tH, d.day,
            Map.add(Map.empty[(Int,Int),Int],
                (d.carid,d.xway),d.tolls))
def writeDExpAnswer(
        hist: TollHist,
        req: DExpReq) =(
            3, req.time, req.carid, req.qid,
            Map.get(Map.get(hist,req.day),
                    (req.carid,req.xway)))
```

4.2 Linear Road: Accident Alerts

While Daily Expenditure requests basically require the system to look up a certain value out of an arbitrarily large dataset, apart from producing the output within the latency time frame, it did not require the system to check events' timing or order. In this regard the Accident Alert requirements are more complex. They can be implemented in three steps: finding stopped cars, computing accident segments and producing the actual alerts. To find out whether a car is stopped, its position from the last four reports must be analyzed. If at least two of the stopped cars share the same position, the segment this position falls in is considered an accident segment. If one of the accident cars first sends a report from another position such that there are not two stopped cars anymore, the accident is said to be cleared. The accident segment however remains valid for the minute the accident was cleared and an alert must be produced for a position report *p*, if there was an accident segment nearby, which was valid in the last minute before *p* was issued.

Figure 4.1 shows this timing on an exemplary accident from a simulation produced by MITSIM: The two cars involved in the accident send reports with IDs 72 and 21 respectively. Both start their trip with a single report from a position before the accident area in segment 44 (car 72 at second 6300 and car 21 on second 6328). The accident starts, when both cars have sent their fourth report from the accident area and ends when the first of them moves again. In this example, the reports showing them moving away from the accident,leaving the expressway and therefore ending the trip are issued 2 seconds after their last position report from the accident's position.

We will see from this example, that even though TeSSLa is designed with a focus on timing properties, when it comes to patterns requiring arbitrarily sized datasets, the presented SBD languages might still be a better fit.

CQL

As mentioned above, we will start with determining which cars have been stopped.

```
CarLocStr:
    SELECT RSTREAM(time,carid,speed,xway,lane,dir,seg,pos)
        FROM CarEventStr[NOW]
```

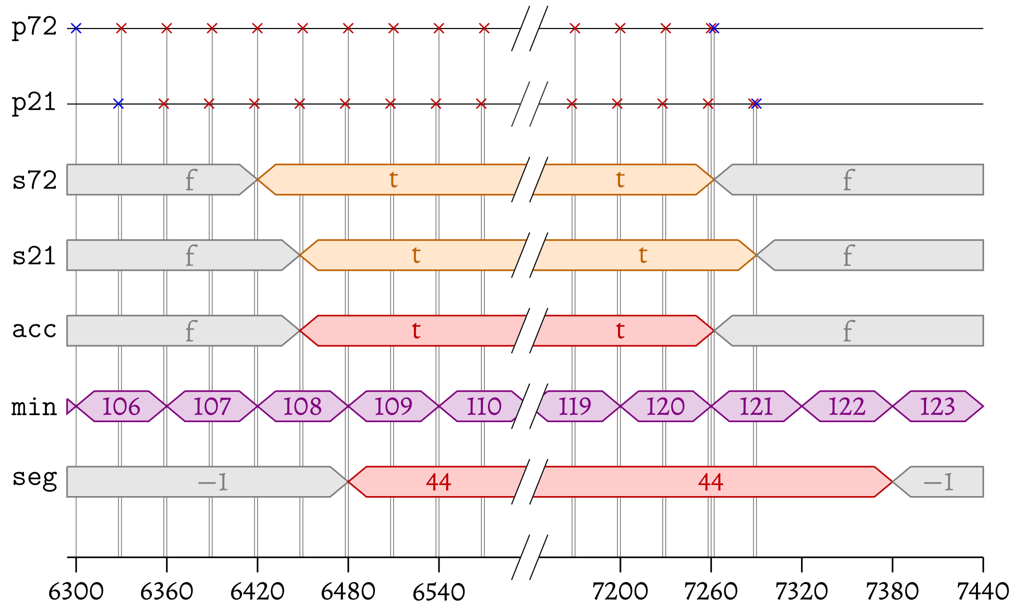


Figure 4.1: Exemplary Accident from Simulation produced by MITSIM. p72 and p21 show position reports of vehicles 72 and 21. The time they are considered stopped is indicated by s72 and s21 and the time of the accident by acc. min shows the minute numbers and seg is the accident segment used for producing alerts with −1 denoting NULL.

```
        WHERE type = 0;

StoppedCars:
    SELECT carid, AVG(xway) AS xway, AVG(lane) AS lane,
           AVG(pos) AS pos, AVG(dir) AS dir, AVG(seg) AS seg
        FROM CarLocStr [PARTITION BY carid ROWS 4]
        GROUP BY carid
        HAVING COUNT DISTINCT (xway,lane,pos,dir) == 1;
```

With the partitioned window we can easily divide the stream of position reports in sub-streams, take the last four position reports of each car and aggregate them. Instead of the AVG of the position we could as well take other aggregates like MIN or MAX since we only consider groups featuring just one distinct position in all of the last four reports. This table however might also contain cars just starting their trip and having therefore just issued a single position report, which naturally only features a single distinct position. To eliminate these cars, the HAVING clause can be extended by AND COUNT (pos) == 4. For an accident there must be at least two cars being stopped at the exact same position:

```
AccSegNow:
    SELECT xway, dir, AVG(seg) AS seg
        FROM StoppedCars
        GROUP BY (xway,lane,pos,dir)
        HAVING COUNT DISTINCT (carid) >= 2;
```

Now we have the current segments with accidents, but we should only output alerts if there was an accident segment valid at the last minute. Getting the timing right is probably the most tricky part here. A straight forward but not quite correct attempt would be to union the current accident segments with the segments removed during the last minute. The problem is, that this would include accidents which just occurred in the current minute so they have not been valid for the last minute. In the example in Figure 4.1 this would lead to alerts from second 6448 where the accident is detected. To ensure they were inserted last minute, we could use the time the entries have been inserted.

```
NewAccStr:
    SELECT RSTREAM(A.xway, A.dir, A.seg, C.time)
        FROM (ISTREAM(AccSegNow)) [NOW] as A, CarLocStr [ROWS 1] as C;
AccSegTime:
    SELECT xway, dir, seg, -1 AS time
        FROM AccSegNow
    UNION
        SELECT xway, dir, seg, -1 AS time
                FROM (DSTREAM(AccSegNow)) [RANGE 1 MINUTE SLIDE 1 MINUTE]
        UNION
        SELECT xway, dir, seg, time
                FROM NewAccStr [RANGE 1 MINUTE];
```

AccSegTime now contains accidents twice if they have been inserted within the last minute and taking the maximum over the time for each accident will either yield the time the accident was detected or -1 if it was detected more than one minute in the past. Thus we can implement the accident segments causing alerts right now as follows:

```
AccSeg:
    SELECT AVG(A.xway), AVG(A.dir), AVG(A.seg)
        FROM AccSegTime AS A,  CarLocStr [ROWS 1] as C
        GROUP BY (xway,dir,seg)
        HAVING FLOOR(MAX(A.time)/60) < FLOOR(AVG(C.time)/60);
```

Finding out which cars recently proceeded to another segment and might approach an area it should be warned for can easily be done with a partitioned window again. Note that a car on the exit lane will end its trip so if the maximum lane number within the last two position reports equals 4, either the last report was from the EXIT lane, or only the second last was from lane 4, which then must have been from a previous trip. In both cases, the vehicle is leaving the expressway or has just started a new trip, it should not receive accident alerts. With the constants EAST = 0 and WEST = 1 we can write

```
NewSegCars:
    SELECT carid
        FROM CarLocStr [PARTITION BY carid ROWS 2]
        GROUP BY carid
        HAVING COUNT DISTINCT (seg) == 2 AND MAX (lane) < 4;
AccNoteTriggger:
```

```
    SELECT carid, time, xway dir, seg
        FROM CarLocStr [NOW]
        WHERE carid IN (SELECT carid FROM NewSegCars);
AccNotifyStr:
    SELECT RSTREAM (1 AS type, T.time, T.carid, A.seg)
        FROM AccSeg AS A, AccNoteTriggger as T
        WHERE (A.xway = T.xway and A.dir = EAST and T.dir = EAST and
              T.seg <= A.seg and T.seg > A.seg - 5)
            OR (A.xway = T.xway and A.dir = WEST and T.dir = WEST and
              T.seg >= A.seg and T.seg < A.seg + 5);
```

For the window function in AccNoteTrigger and the relation-to-stream operator in AccNotifyStr the same thing applies as for the NOW window and RSTREAM operator in ExpOutStr of the previous example implementation.

SQL for PipelineDB

For this example we will be using continuous transforms with output functions which allow us to select certain fields of a stream's events and run a function on each new incoming event. Similarly to the daily expenditure requests we use a continuous transform to take the position reports out of the car_events_stream. But this time, the transform will not just output what it selected, it will directly output the accident alerts we will compute using the function acc_alerts.

```
CREATE OR REPLACE FUNCTION acc_alerts()
  RETURNS trigger AS
  $$
  BEGIN
    CALL upd_reports(NEW);
    CALL upd_stops_accs(NEW);
    RETURN write_alert(NEW);
  END;
  $$
  LANGUAGE plpgsql;

CREATE VIEW acc_alert_stream WITH (action=transform, outputfunc=acc_alerts)
    AS
    SELECT time, carid, xway, lane, dir, seg, pos
    FROM car_event_stream
    WHERE type = 0;
```

In the output function we first add the new report to the most recent reports, a table containing the last four reports of each vehicle

```
CREATE TYPE pos_report AS (
    time INT, carid INT, speed INT, xway INT,
    lane INT, dir   INT, seg   INT, pos  INT
);
```

```
CREATE TABLE last_reports (
    time INT, carid INT, speed INT, xway INT,
    lane INT, dir   INT, seg   INT, pos  INT
);
CREATE TABLE stops (
    carid INT, xway INT, lane INT,
    pos   INT, dir  INT, seg  INT
);
CREATE TABLE accs (
    time INT, xway INT, dir INT, seg  INT, remove BOOLEAN
);
CREATE OR REPLACE PROCEDURE upd_reports(NEW pos_report)
  AS $$
  BEGIN
        INSERT INTO last_reports (
            time, carid, speed, xway,
            lane, dir,   seg,   pos
        ) VALUES (
            NEW.time, NEW.carid, NEW.speed, NEW.xway,
            NEW.lane, NEW.dir,   NEW.seg,   NEW.pos);
    IF (SELECT COUNT(*)
            FROM last_reports
            WHERE carid = NEW.carid) > 4
    THEN
        DELETE FROM last_reports
            WHERE (time,carid) =
                (SELECT time,carid
                    FROM last_reports
                    WHERE carid = NEW.carid
                    ORDER BY time DESC
                    LIMIT 1);
    END IF;
  END;
  $$
  LANGUAGE plpgsql;
```

and after that we update the tables for stops and accidents. This can be done by first checking whether the car is stopped or whether it is moving to take certain action and in the end removing accidents which have been cleared before the last minute.

```
CREATE OR REPLACE PROCEDURE upd_stops_accs(NEW pos_report)
  AS $$
  BEGIN
    IF (SELECT COUNT(*)
            FROM last_reports
            WHERE carid = NEW.carid) > 3
        AND (SELECT COUNT(DISTINCT (xway,lane,pos,dir))
                FROM last_reports
                WHERE carid = NEW.carid) = 1
```

```
    THEN
        CALL stopped(NEW);
    ELSE
        CALL moving(NEW);
    END IF;
    DELETE FROM accs
        WHERE remove
        AND FLOOR(time/60) < FLOOR(NEW.time/60) - 1;
  END;
  $$
  LANGUAGE plpgsql;
```

If the last four reports feature the same position, the car is stopped and we need to check, whether it is already registered as a stopped car. Only if the car was not registered, i.e. the NEW position report is the fourth from the same position, it needs to be registered and checked, if there are other cars stopped at the very same position, to save a new accident segment if it is not present already. If the car was already registered before, these steps have already been done with the position report the car was stopped.

```
CREATE OR REPLACE PROCEDURE stopped(NEW pos_report)
  AS $$
  BEGIN
    IF NOT EXISTS (SELECT * FROM stops
                WHERE carid = NEW.carid
                LIMIT 1)
    THEN
        INSERT INTO stops (
            carid, xway, lane,
            pos,   dir,  seg
        ) VALUES (
            NEW.carid, NEW.xway, NEW.lane,
            NEW.pos,   NEW.dir,  NEW.seg);
        IF (SELECT COUT(DISTINCT carid)
            FROM stops WHERE (xway, lane, pos, dir)=
                (NEW.xway, NEW.lane, NEW.pos, NEW.dir)
            ) = 2 AND
            NOT EXISTS (SELECT *
                        FROM accs
                        WHERE seg = NEW.seg
                            AND NOT remove)
        THEN
            INSERT INTO accs (
                time, xway, dir, seg, remove
            ) VALUES (
                NEW.time, NEW.xway, NEW.dir, NEW.seg, FALSE);
        END IF;
    END IF;
  END;
  $$
```

```
  LANGUAGE plpgsql;
```

If the last four position reports of this car feature more than just a single position, the car is considered moving and therefore, similarly as before, we need to check whether it is registered as stopped car i.e. the car started moving again with the NEW position report. Only if this is the case, the car needs to be unregistered from the stopped cars and it needs to be checked, if this clears an accident at the segment it was stopped and if so, whether it was the only accident at that segment so that the accident segment can be marked for remove.

```
CREATE OR REPLACE PROCEDURE moving(NEW pos_report)
  AS $$
  DECLARE
        stop_seg INT := -1;
  BEGIN
    IF EXISTS (SELECT * FROM stops
                WHERE carid = NEW.carid
                LIMIT 1)
    THEN
        stop_seg = (SELECT seg FROM stops
                    WHERE carid = NEW.carid);
        DELETE FROM stops
            WHERE carid = NEW.carid;
        IF EXISTS (SELECT * FROM accs
                    WHERE seg = stop_seg
                        AND NOT remove
                    LIMIT 1)
            AND (SELECT COUNT(*) FROM stops
                WHERE seg = stop_seg) =
                (SELECT COUNT(DISTINCT (xway,lane,pos,dir))
                    FROM stops)
        THEN
            UPDATE accs
                SET remove = TRUE,
                    time = NEW.time
                WHERE seg = stop_seg;
        END IF;
    END IF;
  END;
  $$
  LANGUAGE plpgsql;
```

Not removing them instantly but just marking them for later removal allows to use the accs table directly for producing alerts with the last position reports, after checking lane, segments and existing accidents in a similar manner as with CQL.

```
CREATE TYPE acc_alert AS (
```

```
    type INT, time INT, emit INT, carid INT,seg INT);

CREATE OR REPLACE FUNCTION write_alert(NEW pos_report)
  RETURNS acc_alert AS
  $$
  DECLARE
    EAST INT := 0;
        WEST INT := 1;
  BEGIN
    IF NEW.lane < 4
        AND (SELECT seg FROM last_reports
                WHERE carid = NEW.carid
                    AND time = NEW.time-30) != NEW.seg
        AND EXISTS (
            SELECT * FROM accs
                WHERE (xway = NEW.xway AND dir = EAST
                        AND NEW.dir = EAST AND NEW.seg <= seg
                        AND NEW.seg > seg - 5)
                    OR (xway = NEW.xway AND dir = WEST
                        AND NEW.dir = WEST AND NEW.seg >= seg
                        AND NEW.seg < seg + 5)
                LIMIT 1)
    THEN
        RETURN (SELECT 1 AS type, NEW.time, CURRENT_TIME AS emit,
                        NEW.carid, seg
                    FROM accs
                    WHERE (xway = NEW.xway AND dir = EAST
                            AND NEW.dir = EAST AND NEW.seg <= seg
                            AND NEW.seg > seg - 5)
                        OR (xway = NEW.xway AND dir = WEST
                            AND NEW.dir = WEST AND NEW.seg >= seg
                            AND NEW.seg < seg + 5));
    ELSE
        RETURN NULL;
    END IF;
  END;
  $$
  LANGUAGE plpgsql;
```

This is just one of multiple possible approaches these properties can be implemented. Instead, one could have used continuous views, more than just one continuous transform, each with less complex output functions, or use user defined aggregates with state values, transition and final function. Many possible approaches also bring a lot of space for optimization: Even though the presented functions try to optimize using techniques like limiting a subquery to just one entry when only the existence of any is of importance, there is still much room for improvement for example by keeping the number of distinct accidents for each segment to avoid the possibly expensive subquery for finding other accidents by the stopped cars if one accident had been removed.

This approach moves from the declarative one with CQL into a procedural direction

while still keeping similarities in tables an queries on them. The CQL approach might be shorter and more elegant in some sense, but a procedural one might be easier to understand by a finer scope on single events instead of windows, especially when it comes to edge cases.

TeSSLa

TeSSLa as a declarative language is used to define streams, but also provides fine control about every single event to, for example, lift some stream's values from one domain into another. Because of this, the approach taken with TeSSLa will show some parallels with the previous for PipelineDB.

First we extract position reports from the input stream omitting unused fields.

```
def latestCarPos = slift1(
    filter(carEventStr, slift1(carEventStr,
                (e: CarEvent) => e.type == 0)),
        (e: CarEvent) => {time =e.time,  carid=e.carid,
                          speed=e.speed, xway =e.xway,
                          lane =e.lane,  dir  =e.dir,
                          seg  =e.seg,   pos  =e.pos})
```

From this we store the latest position of every car.

```
type Position={
    time:   Int, carid:  Int, speed:  Int, xway:   Int,
    lane:   Int, dir:    Int, seg:    Int, pos:    Int}

def carPos: Events[Map[Int,Position]] =
default(
    slift(
        latestCarPos, last(carPos,latestCarPos),
        (p: Position, m: Map[Int,Position]) => Map.add(m,p.carid,p)),
    Map.empty[Int,Position])
```

To find out whether a car is stopping, we need consecutive position reports of it so we augment the position reports in the `latestCarPos` stream with the previous report of the same car.

```
def reportingCar = slift1(latestCarPos,(p: Position)=>p.carid)

def prevCarPos: Events[Position] =
    slift(reportingCar, last(carPos,latestCarPos), lPos)

def lPos(id: Int, pos: Map[Int,Position]) =
        if Map.contains(pos, id) && Map.get(pos, id).lane != 4
```

```
        then Map.get(pos, id)
        else {time=-1,carid=id,speed=-1,xway=-1,
             lane=-1, dir=-1, seg=-1, pos=-1}

def posUpdate =
    slift(prevCarPos, latestCarPos,
          (prev: Position,now: Position) => {p = prev, n = now})
```

`lPos` assumes the last position report to belong to the last trip, if it was sent from an exit lane and therefore also returns UNKNOWN for the last report of the current trip. Now, to check whether a car is stopping, we could store the full previous four position reports as we did in the previous implementation, but in an attempt to keep the amount of data stored in the sets low, we will stick at only considering the last two position reports and count the number of successive records the position remained unchanged.

```
def carStopping: Events[Map[Int,Int]] =
    default(
        slift(last(carStopping,posUpdate), posUpdate, isStopping),
        Map.empty[Int,Int])

type PUpdate= {p: Position, n: Position}

def isStopping(cS: Map[Int,Int], pUD: PUpdate) =
    if samePos(pUD.p,pUD.n)
    then # halting
        if Map.contains(cS,pUD.n.carid)
        then # already registered, increment numReports
            Map.add(cS, pUD.n.carid, Map.get(cS,pUD.n.carid)+1)
        else # add to map
            Map.add(cS,pUD.n.carid,2)
    else # not halting
        if Map.contains(cS,pUD.n.carid)
        then Map.remove(cS,pUD.n.carid)
        else cS

def samePos(a: Position, b: Position) = a.xway == b.xway
    && a.lane == b.lane && a.pos == b.pos && a.dir == b.dir
```

The two conditions of interest now are when a car was stopped i.e. it was “stopping” for the fourth position report and when a car continues moving after being stopped which is when it moves after “stopping” for at least four position reports.

```
type XLPD = {xway: Int, lane: Int, pos: Int, dir: Int}
def stoppedChange: Events[(XLPD, Bool)] =
    slift3(last(carStopping,posUpdate), carStopping, posUpdate,
```

```
          isChanged)

def isChanged(lcS: Map[Int,Int], cS: Map[Int,Int], pUD: PUpdate) =
    if  Map.contains(cS,pUD.n.carid) &&
        Map.get(cS,pUD.n.carid) == 4
    then # 4th pos report at same position
        ({xway = pUD.n.xway, lane = pUD.n.lane,
          pos = pUD.n.pos,   dir  = pUD.n.dir},true)
    else
        if  Map.contains(lcS,pUD.n.carid) &&
            Map.get(lcS,pUD.n.carid) >= 4 &&
            !Map.contains(cS,pUD.n.carid)
        then # stopped car has moved with last position report
            ({xway = pUD.p.xway, lane = pUD.p.lane,
              pos = pUD.p.pos,   dir  = pUD.p.dir},false)
               else # no change
                     ({xway = −1, lane = −1,
                       pos  = −1, dir  = −1},false)

def pureStopChanges =
    filter(stoppedChange,
        slift1(stoppedChange, (e:(XLPD,Bool)) => e._1.lane > 0 &&
           e._1.lane < 4))
```

The stop changes events do not feature the `carid` anymore since we do not need to know which cars stopped to detect an accident, we only need to know the number of cars stopped at the very same position, which we can just count now.

```
def stops: Events[Map[XLPD,Int]] =
    default(
        slift(last(stops,pureStopChanges),
            pureStopChanges, updateStops),
        Map.empty[XLPD,Int])

def updateStops(stps: Map[XLPD,Int],sC: (XLPD,Bool)) =
    if sC._2
    then # new car stopped
        if Map.contains(stps,sC._1)
        then Map.add(stps, sC._1, Map.get(stps,sC._1)+1)
        else Map.add(stps,sC._1,1)
    else # stopped car moved
        if Map.get(stps,sC._1) > 1
        then Map.add(stps,sC._1, Map.get(stps,sC._1)−1)
        else Map.remove(stps,sC._1)
```

With this, we explicitly counted the number of stopped cars at a certain position which we achieved with SQL for PipelineDB by grouping over the position and aggregated via `COUNT` although we could have done it the same way as we did here. After that we proceeded in a similar manner to find accidents, where we already saw that we could have also counted the accidents, in order to omit subqueries when an accident was removed. Here, counting the accidents to know whether there is one left in the same segment, if an accident is removed, follows the idea we used to detect the accidents. Note however, that unlike with SQL, we cannot recheck for accidents that easily because we use maps instead of relations, making us less flexible as we can only *add*, *get* or *remove* entries with certain keys; we will deal with this issue later.

Similar to the event stream for stop changes we proceed now by defining a stream for new or removed accidents, which should contain an event whenever a new stopping car raised the number of cars at that position to 2 or a continuing car reduced it from 2 to 1.

```
def accChangeTrigger =
    filter(pureStopChanges,
        slift(pureStopChanges,
            latestUntil(pureStopChanges, stops),
            isAccChanged))

def isAccChanged(lPSC: (XLPD, Bool),s: Map[XLPD, Int]) =
    if Map.contains(s, lPSC._1)
    then lPSC._2 && Map.get(s,lPSC._1) == 2 ||
          !lPSC._2 && Map.get(s,lPSC._1) == 1
    else false

type XSD = {xway: Int, seg: Int, dir: Int}
def accUpdate: Events[(XSD, Bool)] =
    slift(latestUntil(accChangeTrigger,posUpdate), accChangeTrigger,
        buildUpdate)

def buildUpdate(pUD: PUpdate, lPSC: (XLPD, Bool)) =
        if lPSC._2
        then ({xway = pUD.n.xway, seg = pUD.n.seg,
               dir  = pUD.n.dir}, true)
        else ({xway = pUD.p.xway, seg = pUD.p.seg,
               dir  = pUD.p.dir},false)
```

Now we could use these updates to build up the set of all accidents at a certain moment, either adding or directly removing them as they are detected or get cleared, but accident alerts should be generated even if the accident was cleared within the last minute. Thus we are not only required to update the accidents if there are new ones or some have been cleared, we also need to remove old entries, when a new minute starts. Therefore we merge the `accUpdate` with purge triggers which will basically be created when a new

minute starts and there is something which should probably be removed. The exact conditions will be defined with `removeTrigger` after the accident map.

```
def updANTrigger =
    merge(accUpdate,
        const(({xway = −1, seg = −1, dir = −1},true),
            filter(removeTrigger,removeTrigger)))
```

To be able to check for accidents within a certain region, they should be accessible using the segment on a certain expressway in a given direction. As with the implementation in SQL, a value we need is the time the accident has started and further we need the total number of accidents in the segment.

```
def secNow = slift1(latestCarPos, (p: Position) => p.time)

type accEntry = {time: Int, num: Int}

def accsNow: Events[Map[XSD, accEntry]] =
    default(
        slift4(
            last(accsNow,updANTrigger),
            default(at(updANTrigger,removeNow),List.empty[XSD]),
            at(updANTrigger,secNow),
            updANTrigger,
            updAccsNow),
        Map.empty[XSD, accEntry])
```

The actual updates are performed in the 4-ary function `updAccsNow`. Updating the map by adding a new accident or marking an existing accident as cleared for later removal is quite simple and could be done using the last state of the map, an `accUpdate` and the current second. The difficulty however is to remove old entries: When a new minute begins, all entries for segments should be removed, where the last accident has been cleared more than one minute ago. For a relation in SQL we could just use `DELETE FROM accsNow WHERE num = 0 AND time < segNow-60` but since we are dealing with a map and, even without considering efficiency, TeSSLa currently does not allow to iterate over a map's entry set apart from exploring its key domain, we need to know the exact keys to look at. They will be provided as a list which might only contain entries at the first update of a new minute.

When a new accident is detected, it can just be added to the map as a new entry with the current time or it increments the number of accidents if the segment was already registered, keeping the time it was registered first. When an accident was removed and it was the only accident it needs to be checked, whether a new minute has just started or if the accident was detected at the same second which might happen due to event ordering. In both cases the accident was not valid for a single second of the current minute and should not produce alerts, thus it should be removed directly. If it was not the only

accident, the number can just be decremented. All these actions are done on the purged map where all old entries have been already deleted. If the update trigger did not contain a segment on a valid expressway, the accident map is just purged. Purging is done by recursively traversing the listed entries and removing them if there is no active accident.

```
def updAccsNow(last: Map[XSD, accEntry],remove: List[XSD],
    tm: Int,aUD: (XSD, Bool)): Map[XSD, accEntry] =
    if aUD._1.xway >= 0
    then # not just a purge trigger
        if aUD._2
        then # new accident to add
            if Map.contains(purged,aUD._1)
            then # increment acc number
                Map.add(purged, aUD._1,
                    {time = entry.time,num = entry.num+1})
            else # add new accident
                Map.add(purged,aUD._1,
                    {time = tm, num = 1})
        else # accident removed
            if entry.num == 1 &&
                (entry.time == tm || isNewMinute)
            then
                Map.remove(purged,aUD._1)
            else
                Map.add(purged, aUD._1,
                    {time = entry.time,num = entry.num-1})
    else purged where{ # remove accident
        def entry =
            if Map.contains(purged,aUD._1)
            then Map.get(purged,aUD._1)
            else {time = -1, num = 0}
        def isNewMinute = tm%60 == 0
        def purged = purge(last,remove,0)
    }
def purge(map: Map[XSD, accEntry],remove: List[XSD],
        pos: Int): Map[XSD, accEntry] =
    static if pos < List.size(remove)
    then
        if  Map.contains(map, List.get(remove,pos)) &&
            Map.get(map, List.get(remove,pos)).num == 0
        then
                purge(Map.remove(map, List.get(remove,pos)),
            remove, pos+1)
        else
                purge(map,remove,pos+1)
```

```
    else map
```

To build the list for removal, we start with an empty list every new minute and whenever a last accident in a segment is cleared, it is appended.

```
def markRemove: Events[List[XSD]] =
    default(slift4(
        last(markRemove,updANTrigger),
        default(at(updANTrigger,removeTrigger),false),
        last(accsNow,updANTrigger), updANTrigger,
        updRemoveMarks),
    List.empty[XSD])

def updRemoveMarks(last: List[XSD],newList: Bool,
    lastAccs: Map[XSD,accEntry],upd: (XSD,Bool)): List[XSD] =
    if  !upd._2 && Map.contains(lastAccs,upd._1) &&
        Map.get(lastAccs,upd._1).num == 1
    then List.append(list,upd._1)
    else list where{
        def list = if newList then List.empty[XSD] else last
    }
```

Before the list is replaced by an empty one, it needs to be applied. This can be done by taking a snapshot of the last known state every new minute and we only need to fire a remove trigger, when the list is not empty.

```
def removeNow: Events[List[XSD]] =
    slift(last(markRemove,newMin),removeTrigger,pickList)

def pickList(last: List[XSD], remove: Bool) =
    if remove then last else List.empty[XSD]

def removeTrigger: Events[Bool] =
    slift1(last(markRemove,newMin),
        (mr: List[XSD]) => List.size(mr) > 0)
```

accsNow are defined to contain all accidents valid for the current minute thus, for accident alerts we should take the state after the last minute's last second.

```
def accs : Events [Map[XSD,accEntry]] =
    default(
        last(accsNow,
            filter(minNow,
                slift(last(minNow,minNow),minNow,
                    (lastM: Int, nowM: Int) => lastM != nowM))),
        Map.empty[XSD,accEntry])
```

With this, one can check, whether a position report should trigger an accident alert.

```
def alertTriggers = filter(posUpdate,
    slift(posUpdate, latestUntil(posUpdate, accs), isTriggered))

def isTriggered(pUD: PUpdate, ac: Map[XSD, accEntry]) =
    pUD.p.seg != pUD.n.seg && pUD.n.lane < 4 &&
    nearAcc(ac, {min=m(pUD.n.time), xway=pUD.n.xway,
                 seg=pUD.n.seg,        dir=pUD.n.dir})


type MXSD = {min: Int, xway: Int, seg: Int, dir: Int}

def nearAcc(accs: Map[XSD, accEntry], mxsd: MXSD): Bool =
    Map.contains(accs,
        {xway=mxsd.xway, seg=mxsd.seg       , dir=mxsd.dir}) ||
    Map.contains(accs,
        {xway=mxsd.xway, seg=mxsd.seg+ds    , dir=mxsd.dir}) ||
    Map.contains(accs,
        {xway=mxsd.xway, seg=mxsd.seg+ds*2, dir=mxsd.dir}) ||
    Map.contains(accs,
        {xway=mxsd.xway, seg=mxsd.seg+ds*3, dir=mxsd.dir}) ||
    Map.contains(accs,
        {xway=mxsd.xway, seg=mxsd.seg+ds*4, dir=mxsd.dir}) where{
            def ds = dirSign(mxsd.dir)}

def dirSign(dir: Int): Int = if dir == 0 then 1 else -1
```

To finally write the accident alerts, we recall that there might be more than one accident segment such that one position report triggers more than one accident alert. In comparison to the other languages, for TeSSLa this is an issue since it does not allow multiple events of the same timestamp within one stream. One way to face it would be to slightly delay further accident alerts to serialize them in a similar way as we do for input stream events by applying a finer granulated time domain and use the time they are inputted into the interpreter as TeSSLa timestamp. However, this combination will only work, if the steps between input timestamps are always big enough to fit up to 5 accident alerts after each other without overlapping possible alerts for a subsequent report. To output them all at the same timestamp of the triggering position report instead, we can aggregate them for example in a set and disassemble them in the post processing step we need anyway to add the emit time.

```
def accidentAlert: Events[(Int, Int, Int, Set[Int])] =
    slift(alertTriggers, latestUntil(alertTriggers, accs),
            writeAccAlert)
```

```
def writeAccAlert(pUD: PUpdate, ac: Map[XSD, accEntry]) =
        (1, pUD.n.time, pUD.n.carid, accSegs(
                        ac,{min=m(pUD.n.time),xway=pUD.n.xway,
                            seg=pUD.n.seg,       dir=pUD.n.dir}))

def accSegs(ac: Map[XSD, accEntry],mxsd: MXSD): Set[Int] =
    addAccSeg(ac,mxsd,4,addAccSeg(
      ac,mxsd,3,addAccSeg(
        ac,mxsd,2,addAccSeg(
          ac,mxsd,1,addAccSeg(
            ac,mxsd,0,Set.empty[Int])))))

def addAccSeg(ac: Map[XSD, accEntry],mxsd: MXSD,
              off: Int, accSegs: Set[Int]): Set[Int] =
    if Map.contains(ac,{xway=mxsd.xway, seg=checkSeg, dir=mxsd.dir})
    then Set.add(accSegs,checkSeg)
    else accSegs where {
        def checkSeg = mxsd.seg+dirSign(mxsd.dir)*off}
```

4.3 Burst Pattern Example

This property can be implemented straight forward with TeSSLa, not only because of handy inbuilts, but also because of its view on time and timing: An event's time is directly taken from the input and all events are assumed to be globally ordered by these timestamps for all operations. For other languages in practical use, we cannot count on such properties. First of all PipelineDB's streams indeed always contain a column `arrival_time`, but rather than the time it is *read* by the system, it shows the *transaction* time. Thus e.g. if the stream events are read from a pipe in a long transaction they will all get the same timestamp value, which is obviously useless for comparing time of events. To counter this, one could apply a continuous transform and use one of the inbuilt time functions to give the events an appropriate stamp. Since TeSSLa reads the timestamps along the values if the trace is input from a file as we will use for this example, we will take this time value as an extra column in SQL or CQL, ignoring system time values as the `arrival_time` column.

Further, for the relational languages, there is no guarantee that operators work synchronized on the input tuple's times. For CQL, since we are not considering a concrete implementation, we will simply assume that the system can catch up and work synchronized with the input, meaning for example, that if we join the `NOW` windows of two different streams, we will always receive a tuple iff the two input stream's events featured the same timestamp. For SQL however, we will examine different attempts to ensure PipelineDB operates as expected.

CQL

We begin by checking the silencing condition for which we just need to compare the last known values of the a and b streams.

```
SilenceCond:
    SELECT la.a > lb.b AS c, MAX(la.time,lb.time) AS time
        FROM aStream [ROWS 1] AS la, bStream [ROWS 1] AS lb;
```

The silencing condition demands the eStream to be silent if its value is TRUE, thus the properties of the silencing conditions are met if either its value equals FALSE or it has been changed after the last e event appeared.

```
SilenceChange:
    SELECT nowC.c, nowC.time AS change
        FROM DSTREAM(SilenceCond) [ROWS 1] AS lastC, SilenceCond AS nowC
        WHERE lastC.c != nowC.c;

SilenceOK:
    SELECT !c OR lc.time > le.time AS sil
        FROM SilenceChange, eStream [ROWS 1];
```

Since we do not consider events for bursts if the stream should be silenced, for checking the actual bursts we can remove all events issued for the eStream while the silence condition is TRUE. This also ensures events to be ignored if the silencing condition is not yet known.

```
BurstEventStr:
    SELECT RSTREAM(e)
        FROM eStream [NOW], IsSilenced
        WHERE !c;
```

Now we have to somehow detect burst starts. An event in eStream starts a new burst if it is the first event after the silencing condition was changed to FALSE or if the last burst is over. Therefore, additionally to the BurstEventStr and the changes to the silencing condition, the start of the previous burst is required which might be solved using recursion.

```
RecBurstStart:
    SELECT CASE WHEN MAX(start) + BURST_LENGTH + WAITING_PERIOD > MAX(time)
                OR MAX(change) <= MAX(start)
            THEN MAX(start) ELSE MAX(time) END AS start
        FROM RecBurstStart, BurstEventStr [ROWS 1], SilenceChange
    UNION
    SELECT -1 AS start;

BurstStart:
    SELECT MAX(start)
        FROM RecBurstStart;
```

With this we can check the burst properties by just counting the events since the last start and checking whether all events belonging to bursts appear within the time frame of BURST_LENGTH after the the burst start.

```
AmountOK:
    SELECT COUNT(time) <= BURST_AMOUNT AS num
        FROM eStream [RANGE BURST_LENGTH], BurstStart
        WHERE time >= start;

WaitingOK:
    SELECT time <= start + BURST_LENGTH AS wait
        FROM eStream [ROWS 1], BurstStart;
```

The silencing part and the two burst parts must all apply for the burst pattern example property which has to be outputted whenever its value changes.

```
PropertyOK:
    SELECT sil AND num AND wait AS p
        FROM SilenceOK, AmountOK, WaitingOK

PropertyStr:
    SELECT RSTREAM(nowVal.p)
        FROM DSTREAM(PropertyOK) [ROWS 1] AS lastVal, PropertyOK AS nowVal
        WHERE lastVal.p != nowVal.p
```

SQL for PipelineDB

As mentioned above, there are multiple ways to implement properties like these with PipelineDB showing different behavior: Some might yield incomplete or wrong results if the operators do not work synchronously, others will eventually get the output right and finally it can give correct output right away. Incomplete or wrong results might be produced if the input is processed in different substreams independently like evaluating the silencing condition and writing it to some table so it can be accessed by other continuous transforms. In this example, whenever there is an e event, we could use such a table containing the last value of the silencing condition to check whether it belongs to some burst or if it appeared during a silenced phase. If however the value of the silencing condition will change at the same time and is not yet up to date e.g. due to input event ordering, PipelineDB will falsely output an error or incorrectly classify as burst event and therefore falsely not output an error. Furthermore, examples like this might be implemented using continuous views and defining hierarchical views on them. Dependent on how final views are used to generate streaming output, it might be wrong or incomplete as well, but if the continuous views do not drop events, the final views for output will contain the right answers when all continuous views have been, up to a certain timestep, populated completely.

To ensure that all correct results are generated when the events of a single timestep are processed, they might be computed over a single stream where an event contains all

values of relevant stream's events for that timestamp. A stream like that could be implemented by collecting all input stream's events until a new timestamp appears. Then an event containing all aggregated values with NULL values for streams with no event at that timestep is outputted. Using this approach, the burst pattern example can be implemented relatively simple by using functions and procedures similarly as we implemented Linear Road's accident alerts. Instead of repeating this attempt, we will show how it can be done with hierarchical views despite the disadvantages we will later examine a little further, in order to give a brief impression on how limited SQL without UDAs or trigger functions is on timing and ordering properties like this.

First of all, we define the input as a single stream containing the values of all events for a single timestamp. Although it could be generated from individual streams as described above, we will skip this step here for simplicity.

```
CREATE FOREIGN TABLE in_stream (
        time bigint, a bigint, b bigint, e_u text)
SERVER pipelinedb;
```

The e events could just be of any type to be distinguished from NULL, here and for the implementation we will use strings. Now we will use a single continuous view, as an append only table on which we will check for the burst pattern.

```
CREATE VIEW burst_trace WITH (action=materialize) AS
       SELECT time, a, b, e_u AS e
       FROM in_stream;
```

To compute the value of the silencing condition with CQL we just used the last values of "a" and "b" respectively by applying a tuple based window. But since we work on the stream as on a constantly growing table, we need to define all properties as if the table was completely filled with all events but in a non-blocking way to enable generating output up to the last known event while incrementally processing the input. For the silencing condition, this means that we need to combine each of the "a" values with the corresponding last "b" value and vice versa since every new value in "a" as in "b" sets the silencing condition's new value.

```
CREATE VIEW a_vals AS
       SELECT time, a
       FROM burst_trace
       WHERE a IS NOT NULL;
CREATE VIEW b_vals AS
       SELECT time, b
       FROM burst_trace
       WHERE b IS NOT NULL;
CREATE VIEW a_latest_b AS
       SELECT avs.time, a, b
       FROM a_vals AS avs, b_vals AS bvs
       WHERE avs.time >= bvs.time
```

```
       AND NOT EXISTS (SELECT time
                        FROM b_vals
                        WHERE avs.time >= time
                        AND time > bvs.time);
CREATE VIEW b_latest_a AS
       SELECT bvs.time, a, b
       FROM a_vals AS avs, b_vals AS bvs
       WHERE bvs.time >= avs.time
       AND NOT EXISTS (SELECT time
                        FROM a_vals
                        WHERE bvs.time >= time
                        AND time > avs.time);
```

The NOT EXISTS can be used here in a non-blocking way since no later event will feature a timestamp less or equal the timestamp of an earlier event. The same way ORDER BY time DESC LIMIT 1 could be used here but this could not be applied to a continuous view directly. Using this, the silencing condition can be defined in a straightforward way.

```
CREATE VIEW c AS
       SELECT time, CASE WHEN a > b THEN 'true' ELSE 'false' END AS val
       FROM (SELECT * FROM a_latest_b
       UNION SELECT * FROM b_latest_a) AS ab_vals;
```

Next we take all e events and only preserve events where the most recent value of the silencing condition is FALSE i.e. the value is known and not silencing.

```
CREATE VIEW raw_burst_events AS
       SELECT time, e
       FROM burst_trace
       WHERE e IS NOT NULL;
CREATE VIEW c_false AS
       SELECT time AS cf_time
       FROM c
       WHERE val = 'false';
CREATE VIEW c_true AS
       SELECT time AS ct_time
       FROM c
       WHERE val = 'true';
CREATE VIEW burst_events AS
       SELECT time, e
       FROM raw_burst_events
       WHERE EXISTS(SELECT cf_time
            FROM c_false
            WHERE cf_time <= time
                AND NOT EXISTS(
                     SELECT ct_time
                     FROM c_true
```

```
                    WHERE cf_time <= ct_time
                    AND ct_time <= time));
```

With this, violations of the silencing property are the `raw_burst_events` not included in `burst_events`.

```
CREATE VIEW no_event_violations AS
       SELECT time
       FROM raw_burst_events
       WHERE time NOT IN (SELECT time
                          FROM burst_events);
```

For inspecting the actual bursts, we proceed finding the times when the silencing condition falls from TRUE to FALSE indicating the start of a new burst interval.

```
CREATE VIEW c_falls AS
       SELECT cf_time AS fall
       FROM c_false
       WHERE (SELECT val FROM c
              WHERE time < cf_time
              ORDER BY time DESC LIMIT 1);
```

By matching each burst event with the corresponding last `fall` time, we divide them into the burst intervals, of which the first burst event always starts a new burst.

```
CREATE VIEW burst_intervals AS
       SELECT be.time, cf.fall
       FROM burst_events be, c_falls cf
       WHERE cf.fall = (SELECT fall FROM c_falls
                        WHERE fall <= be.time
                        ORDER BY fall DESC LIMIT 1);
CREATE VIEW first_starts AS
       SELECT MIN(time)
       FROM burst_intervals
       GROUP BY fall;
```

Now, as with CQL, we recursively find the next burst starts. To simplify the recursion, we first find the preceding burst events for each burst within the same burst interval.

```
CREATE VIEW time_last AS
       SELECT t.time AS ttime, l.time AS ltime
       FROM burst_intervals t, burst_intervals l
       WHERE l.fall = t.fall
             AND l.time = (SELECT time
                           FROM burst_interval
                           WHERE time < t.time
                           ORDER BY time DESC
                           LIMIT 1);
```

With this we can not only find the next burst starts recursively, we can also match each burst event with a certain burst and find the number of each burst event within its burst.

```
CREATE VIEW burst_starts AS
       WITH RECURSIVE starts AS (
            SELECT time AS stime, time AS start, 1 AS event_count
            FROM first_starts
       UNION
            SELECT time AS stime,
                 CASE WHEN time - start < 3000
                 THEN start ELSE time END AS start,
                 CASE WHEN time - start < 3000
                 THEN event_count + 1 ELSE 1 END AS event_count
            FROM starts,burst_events,time_last
            WHERE time = ttime AND stime = ltime
       )
       SELECT stime AS time, start, event_count
       FROM starts;
```

This makes finding burst property violations very easy since we only need to check, if the number of a certain event within its burst exceeds the burst amount of 3 or whether the start of its burst lies more than the burst length of 2 seconds in the past.

```
CREATE VIEW burst_violations AS
       SELECT time
       FROM burst_starts
       WHERE event_count > 3
       OR time - start >= 2000;
```

Together with the violations of the silencing condition, they make up the set of all burst events for which the burst property we want to check is FALSE.

```
CREATE VIEW p AS
       SELECT time, CASE WHEN time IN (SELECT time FROM violations)
       THEN 'false' ELSE 'true' END AS val
       FROM burst_trace;
```

As this property should only to appear in the output when its value changes, the output can be implemented with the following view.

```
CREATE VIEW pure_p AS
       SELECT time, val
       FROM p p1
       WHERE val IS DISTINCT FROM
           (SELECT val FROM p p2
            WHERE p2.time < p1.time
            ORDER BY time DESC LIMIT 1);
```

This implementation uses many expensive joins and subqueries such as to match some events with preceding events which especially shows an advantage of processing input rows one after another over viewing them as a list of all. Apart from that there are many ways to optimize these views, this whole approach of hierarchical views on append only continuous views is poorly suited for long running continuous systems as the over time increasing sizes of input tables will constantly decrease performance. Note also, that our output view is no continuous view since it does not select from a stream. Therefore it is not associated with an output stream making continuous output as a stream more complicated.

There might be applications, where this approach is particularly handy, for the properties can be specified easily using an SQL query and the input appears in a few bulks rather than a long continuous stream. On top of this, it should be possible to safely drop old input tuples for keeping performance at a reasonable level. For this burst pattern example however, the approach of handling input events one after another is much better suited as it allows to reduce the growing memory requirements for longer runs to a constant amount of saved values and is more convenient to implement.

TeSSLa

As advertised earlier, the implementation in TeSSLa taken from [21] is very short. Defining input, intermediate streams and output only takes a few lines:

```
in a: Events[Int]
in b: Events[Int]
in e: Events[Unit]

# Specify silencing condition
def c := a > b

# Specify correctness property
def p := if c
         then noEvent(e, since = rising(c))
         else burstsSince(e, burstLength = 2s,
                             waitingPeriod = 1s,
                             burstAmount = 3,
                             since = falling(c))

# Output
out a # signal
out b # signal
out e # unit events
out pure(p) as p # signal
```

It makes use of standard library functions supported by TeSSLa's interpreter and compiler. `rising` and `falling` take streams of boolean values and describe unit streams

with events whenever the input stream's signal value changes from `false` to `true` or vice versa respectively. In the standard library they are defined from basic operators as follows:

```
def unitIf(cond: Events[Bool]): Events[Unit] = constIf((), cond)

def prev[A](a: Events[A]): Events[A] = last(a,a)

def rising(condition: Events[Bool]): Events[Unit] =
  unitIf(condition && !prev(condition))

def falling(condition: Events[Bool]): Events[Unit] =
  unitIf(!condition && prev(condition))
```

`noEvent` is a boolean signal taking the value `true` if there was no event in the first input stream since the last reset given by an event in the second input stream. Again from the standard library:

```
def resetCount[A,B](events: Events[A], reset: Events[B]): Events[Int] =
  count where {
    def count: Events[Int] = default(
      # `reset` contains the latest event
      if default(time(reset) > time(events), false)
      then 0
      # `reset` and `events` latest event happen simultaneously
      else if default(time(reset) == time(events), false)
      then 1
      # `events` contains the latest event > increment counter
      else last(count, events) + 1,
      0)
}

def noEvent[A,B](on: Events[A], since: Events[B]): Events[Bool] =
  resetCount(on, reset = since) == 0
```

The actual bursts are checked by using the function `burstsSince` which counts the events from the input stream, if the counter, which is reset whenever a new burst starts, reaches the burst amount or there is an event later than burst length from the last burst start it will take the value `false`. Else and by default the pattern's conditions are met and the value is `true`. The standard library suggests the following definition:

```
def on[A,B](trigger: Events[A], stream: Events[B]): Events[B] =
  filter(first(stream, trigger), time(trigger) >= time(stream))

def first[T, U](stream1: Events[T], stream2: Events[U]): Events[T] =
```

```
  slift(stream1, stream2, (x: T, _: U) => x)

def burstsSince[A,B](e: Events[A], burstLength: Int, waitingPeriod:
    Int, burstAmount: Int, since: Events[B]): Events[Bool] = {
  def burstStarts: Events[A] =
    defaultFrom(
      filter(e, last(time(burstStarts), e) < on(e, time(since)) ||
        time(e) - last(time(burstStarts), e) >= burstLength +
            waitingPeriod),
      e)
  resetCount(e, reset = burstStarts) <= burstAmount &&
    default(time(e) < time(burstStarts) + burstLength, true)
}
```

Finally to only output the property's value if it changes, the standard function `pure()` is used.

```
def pure[T](x: Events[T]): Events[T] =
  filter(x, merge(last(x,x) != x, true))
```

Without using standard library functions with TeSSLa, a tuple-wise incremental implementation for PipelineDB is not much more intricate and might even feel more convenient to someone used to classical databases. However, as long as the number of values to store is assessable, concepts like signals and signal lift in TeSSLa are relatively easy to use and might, in comparison to SQL, save numerous `INSERT` and `SELECT` statements. Building functions, methods or custom aggregates for reuse is possible for PipelineDB however libraries in TeSSLa integrate more naturally into the set of basic operators, allowing to even replace them completely making TeSSLa exceptionally easy to use in application domains with a comprehensive specific library.

4.4 Property Implementation Summary

Linear Road: Daily Expenditures

The challenge in daily expenditures was to store a historical dataset and access it for query answers which has simple implementations especially for the SBD languages. In CQL we could just apply an unbounded window to the stream of historical data and join the resulting table with the queries. For PipelineDB a continuous view does essentially the same thing as CQL's unbounded window and the join with the queries can be implemented using a continuous transform. For TeSSLa we used maps to save the historical data. Then we lifted the stream of queries to the query answers by addressing the desired data from the latest state of the historical dataset, using the request information as key. Building up the history here by successively updating the latest state was a bit more complicated than for the SBD languages where we could just use inbuilt functionality.

Linear Road: Accident Alerts

For the accident alerts, the partitioned window in CQL was a handy tool. To determine, whether the preconditions of a position report to trigger an accident alert are met, we could use the partitioned window for the last two reports per `carid`. To find the stopped cars, we could just partition the reports by `carid` and if the last four reports of a car only contain a single distinct position, it is stopped. By grouping the relation of stopped cars by position, groups with at least two different stopped cars are the accidents. The difficulty of getting the timing right for alerts to be generated, was handled by adding the detection time to new accidents and union them with the current minutes `DSTREAM` of the accidents. This table could then be used by joining with the position reports for which the accident alert preconditions are met, and considering position as well as start time of the accidents.

In SQL for PipelineDB this was implemented using a single continuous transform with all complexity in the output function. It allowed to procedurally handle incoming stream events one after another. A new row was first used to update the last position reports of each cars, essentially the same table as we obtained in CQL using the partitioned window on `carid` with four rows. Then the lists of stopped cars and accidents respectively are updated if needed and finally used to write alerts. Specifying the system's behavior in a procedural way using `INSERT`, `SELECT` and `DELETE` statements endows, but also imposes performance control on the user. Furthermore, handling events one after another as they arrive rather than keeping the view on windows, might be easier for timing patterns, especially considering edge cases.

Using TeSSLa, we started the implementation by saving the last known position for each car in a map and used it to lift new position reports to position updates, containing the last report's information besides the recent. Then, if both shared the same position, we used another map to count the number of reports a car sent without changing its position. If this number is raised to 4, the car is considered stopped and was added using its position as key to a map of stopped cars. Similarly, if the number of stopped cars at a certain position reaches 2, the segment is added to the map of the current minute's accidents. Just before a new minute starts, we take a snapshot of this map to write accident alerts for the following minute. The difficulty here was, to keep all accidents of a minute until the moment we can take this snapshot and remove old entries afterwards. In contrast to SQL with PipelineDB we cannot simply remove old entries similarly to a "`DELETE...WHERE`" since we use a map and can only add, get and remove entries for certain keys. Therefore we built up a list containing all entries which should be removed and applied it at new minute's starts.

Burst Pattern Example

To implement the burst pattern example in CQL, we started by comparing the last known "`a`" and "`b`" values by joining windows of single rows to evaluate the silencing condition. Since new bursts not only start whenever the silencing condition has just changed to `false`, but also after the last burst is over, which depends on the previous burst start, we used a recursive self-join to determine them. Together with the silencing condition and

the "e" events, the burst starts could then be used to decide whether the burst property is met or violated.

For SQL we first considered different approaches how the burst pattern could be implemented, and decided to present an implementation based on a single continuous view as it shows, how to implement such continuous properties using SQL with a view on the whole table of stream events – in contrast to CQL where we used windows to only focus on the last burst. We also noted, that this approach is hardly practical and one should use UDAs and/or continuous transforms instead. We did not present these approaches since their key ideas have already been covered with the accident alert implementation. Working with the said continuous view, we saw how inconvenient it is for evaluating ordering properties by matching events with the corresponding latest previous events. Again, to find the burst starts we used a recursive view.

For TeSSLa we presented the short specification using standard library functions and further showed how they can be implemented using basic operators only. We argued, that an implementation for PipelineDB could compete in terms of simplicity as it might be more convenient to use considering relational algebra and SQL to be commonly known, assuming the standard or a specific library for TeSSLa not to be given.

5

SRV Implementation with PipelineDB

Stream Runtime Verification, as continuously deriving output streams from the incoming events to check a program trace for certain criteria is, as one might state, just regular data stream processing, a task which PipelineDB was specifically designed for. Thus the first question to answer here is, what this implementation should accomplish in particular. Therefore we take a look at what is supported by the TeSSLa interpreter and how it can be used: It can, as described in the tutorial [18], directly listen to events of a program from instrumented C source code, running in parallel. Also it supports a special trace syntax for input events. As with the program events, the trace can be read from regular files as offline monitoring or through a named fifo as online monitoring. A TeSSLa specification file defines the output streams to be derived from the input.

The implementation will focus on the TeSSLa trace syntax, which might be processed in an online or offline manner. A single specification file in SQL should be sufficient to define in- and output stream behavior and the output should be observable in real time for online scenarios. The complete behavior might indeed be implemented in a single file in SQL for PipelineDB, but this would always require a lot of extra work e.g. for parsing the input, possibly outshining the actual stream property specification. On top of that, the probably most straightforward way of reading continuous input as a long running `COPY stream_name FROM 'named_fifo'` has the disadvantage of being a single transaction, giving all read tuples the same `arrival_timestamp`, therefore bypassing any use of the inbuilt time-based windows and significantly limiting specification potential with continuous views. Therefore the implementation should insert events one after another at the time they arrive.

5.1 Implemented Program Structure

Similar to the TeSSLa interpreter, the implementation reads the input trace from a file, which might be a named fifo for online monitoring. The implementation is split into two parts, an input parser and a PipelineDB communicator. The input parser reads the input trace, translates it into events which can directly be inserted into the SBD and passes them to the communicator through a named fifo. For examining the effect of system's latency on the output like output event ordering, the parser also allows to delay input

events read for offline monitoring to simulate the delays of events as if the monitoring was performed online. The PipelineDB communicator will initialize the stream evaluation by passing the specification file to the SBD. Then, whenever it reads a new event from the input parser, it will directly pass it to the database. The output stream can be observed using the `srv_out` fifo which will be opened by the communicator. It might be written by the communicator itself as well as by calls from PipelineDB for example in trigger functions. The output of the communicator must be prepared by predefining queries in the SQL specification and listing the names of them in the end of the specification file to be parsed by the communicator. Each of these queries corresponds to one output stream and takes a time value as the last known event's time within this stream, to simulate an insert stream behavior by successively polling events with later timestamps. If the output should be done by the SBD exclusively, the list of queries can be left empty and the communicator will not perform any polling. An overview of the structure can be examined in Figure 5.1. The whole implementation can be run in a docker container including the PipelineDB server.

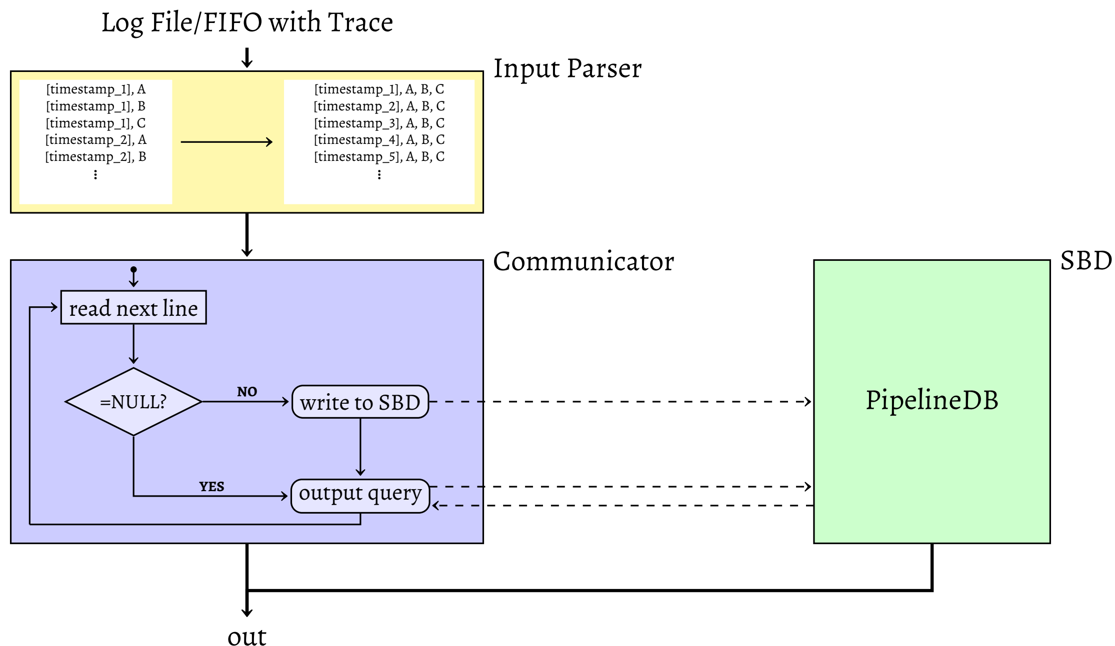


Figure 5.1: Implemented Program Structure for SRV with PipelineDB

5.2 SQL Specification File

Since, as outlined earlier, for timing and ordering patterns it might be necessary to handle all events of the different streams for a single timestamp as one composed event, for simplicity we will only define a single PipelineDB input stream. The read input events will, as described in 4.3, be collected by the input parser until a new timestamp is read or the end of the input is detected. Then they are used to compose an event to be inserted

into the PipelineDB input stream. This stream can be defined in the SQL specification file as follows:

```
CREATE FOREIGN TABLE in_stream (
           time bigint, e_1 type1, e_2 type_2, ..., e_n type_n)
SERVER pipelinedb;

PREPARE insert (bigint,type1,type_2,...,type_n) AS
        INSERT INTO in_stream VALUES($1, $2, $3, ..., $n+1);
```

where e_1 to e_n are the different streams found in the trace log with the corresponding event types type_1 to type_n. Since unit events do not carry a specific value but should be distinct from NULL if it is present, the parser assumes unit fields in the input stream specification to be of text type. Since the type text however can also be used for TeSSLa streams with String events, another hint for the input parser, which is also reading the specification file to know which events to look for, is needed to tell which events are of type unit. Therefore the stream name is just extended as name_u so that whenever there is an event at some timestamp for stream name in the trace, the parser will write 'unit' instead of NULL for the SBD. An example of this can be seen in the burst pattern example implementation for PipelineDB, Chapter 4.3, where the input stream e is of type unit. Actual String typed event values are written as "string" in TeSSLa's trace syntax and since PipelineDB needs a text to be written as 'text' for insertions, the parser will replace leading and ending double quotation marks of input event's values by the single quotation marks. All other event values are passed as they are for the corresponding field in an input row, which works well for numeric and boolean values.

On the in_stream, all of PipelineDB's features can be utilized, to implement the desired behavior where the output can either be directly written using calls like COPY (query) TO '/tmp/srv_out'; or selected from tables or views with prepared queries like

```
PREPARE stream_out (bigint) AS
        SELECT time, ': stream_out = ' || val
        FROM stream_out_vals
        WHERE time > $1;

/* LIST OF OUTPUT QUERIES marked by --> !list! */
-- stream_out
```

The list of output queries should end the specification file.

Example Specification File

To keep it short, but still provide a full specification, we use the example of the burst pattern and add the additional parts of the specification to the already presented implementation in 4.3 to complete a working specification:

```
/* CLEAN TABLES */
DROP FOREIGN TABLE IF EXISTS in_stream CASCADE;

/*
 * Implementation as already presented
 */

 /* PREPARE FOR INPUT */
PREPARE insert (bigint,bigint,bigint,text) AS
        INSERT INTO in_stream VALUES($1, $2, $3, $4);

/* PREPARE FOR OUTPUT (as Istream) */
PREPARE a_out (bigint) AS
        SELECT time, ':␣a␣=␣' || a
        FROM a_vals
        WHERE time > $1;
PREPARE b_out (bigint) AS
        SELECT time, ':␣b␣=␣' || b
        FROM b_vals
        WHERE time > $1;
PREPARE e_out (bigint) AS
        SELECT time, ':␣e␣=␣()'
        FROM burst_events_nofilter
        WHERE time > $1;
PREPARE p_out (bigint) AS
        SELECT time, ':␣p␣=␣' || val
        FROM pure_p
        WHERE time > $1;

/* LIST OF OUTPUT QUERIES marked by --> !list! */
-- a_out
-- b_out
-- p_out
-- e_out
```

The cleaning by dropping all tables and views can be omitted for the docker implementation but should be considered if ran on a PipelineDB server which is not restarted between runs.

5.3 SRV PipelineDB Summary

The described implementation allows to perform Stream Runtime Verification in a similar way as with the TeSSLa interpreter on traces in TeSSLa's trace syntax with primitive input event types. The specification is defined in an SQL specification file analogous to the TeSSLa specification file for the TeSSLa interpreter. We have seen, that for example approaches like hierarchical views might lead to poorly efficient implementations, but

other attempts such as handling events one after another might show competitive efficiency, which in particular has not been evaluated. Further, the implementations must pay attention to the order of processing input events and outputting events, otherwise expensively evaluated events might appear in the output long after more simple events of later timestamps or the output might even not be complete or contain wrong results.

6

Linear Road Implementation with TeSSLa

Other than for the SRV implementation with PipelineDB, for this implementation input and output is strictly defined by the benchmark. Although the Benchmark describes travel time estimation queries in detail, for the implementations from the benchmark's original paper [2] they left them out, the validator does not check them, MITSIM does not provide segment histories and the requests it generates always carry zeros for S_{init}, S_{end}, DOW and TOD. Thus we will also leave out the travel time estimations and ignore all such requests.

6.1 Program Structure

As system input, MITSIM generates two CSV files, one containing the toll history and one for the stream events. The implementation will start the TeSSLa interpreter in parallel and first of all initialize the benchmark by reading the toll history and feeding it into the interpreter. The communication between the benchmark driver implementation and the TeSSLa interpreter is implemented using TeSSLa's trace syntax via named fifos.

After the historical data is passed, the simulation starts at second 0 by reading the lines from the stream data input file, which are ordered by the time field, checking the time of the event and if it is not later than the current time from the beginning of the simulation, it is written to the TeSSLa input fifo. If an event with a later time value is read, the thread will sleep until it can pass it to TeSSLa. Another thread running in parallel is permanently listening for TeSSLa output events. If it reads a new event from TeSSLa, it adds the time which has passed since the beginning of the simulation as emit time and writes the values to the corresponding output files. If an accident alert event from TeSSLa contains a set of multiple accident segments, it will separately write a line for each segment to the accident alert output file. The output files are then taken by the validator together with the input files to check the system output for correctness. The program structure is illustrated in Figure 6.1. In parallel to communicating with the TeSSLa interpreter, the benchmark driver collects system statistics about used memory, used memory by the java virtual machine the interpreter runs in and the CPU utilization of the JVM. The implementation can be run in a docker container including plotting system and latency statistics and validating the system output after the benchmark is finished.

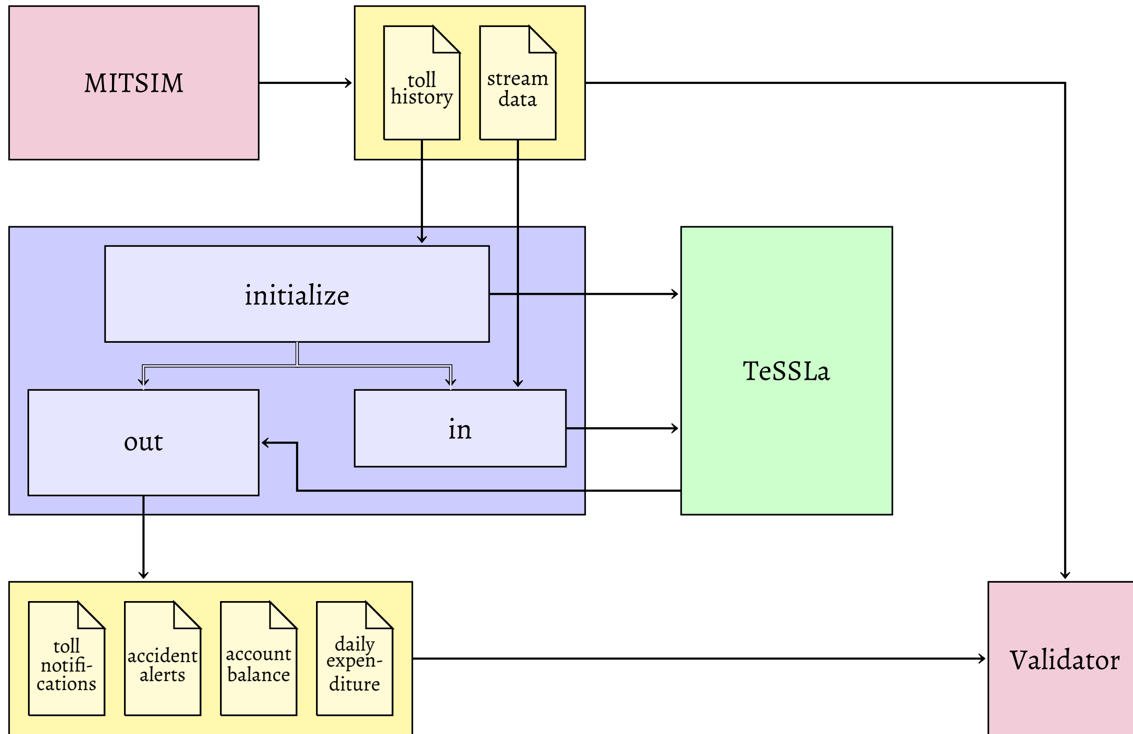


Figure 6.1: Program Structure for the Linear Road Benchmark Implementation with TeSSLa

6.2 Notes on Correctness

Since there is the validator, checking the system's output should be fairly easy and for all simulations presented in the following the validator approved the answers except for the 10 expressway simulation. A closer look into the discrepancies allowed to extract minimal problematic subsets of the input and to identify the following errors.

The simulation contained two different accidents at the very same expressway, segment, lane and position. While the TeSSLa implementation correctly identified their starts and ends, the validator treated them as a single accident with the start of the first and the end of the second accident resulting in wrong accident alerts between the two accidents. Furtermore the simulation contained two accidents which were cleared in the beginning of a new minute, more precisely on 2nd and 6th second of that minute. Following the specification (See Table 3.3) the accident still holds for that minute so there should be alerts in the following minute too which the TeSSLa implementation issued correctly yet the validator did not. A very short accident which was already cleared one minute after it started was not found by the validator at all. Since different handling of accidents leads to different results for the tolls, these errors also affected some toll alerts and account balance answers.

Another error with toll alerts appeared in two situations where the lane average velocity calculated by the TeSSLa implementation was one mile per hour above the validator's result: Indeed the correct average value before rounding down was very close to

the next higher whole number, but the error of the TeSSLa version was not due to precision issues. The implementation simply assumes there cannot be more than two position reports for a single car in one minute which is reasonable since the specification of the benchmark states that each car sends its reports exatly every 30 seconds; if the time between two reports with different positions is smaller, the *No Position Interpolation* assumption also defined in the specification cannot hold. Nevertheless when it comes to accidents, the simulations generated by MITSIM often contain two subsequent reports within less than 30 seconds which, in these edge cases, caused the wrong Lav.

Therefore, since all negative results of the validator for the 10 expressway simulation are either caused by errors of the validator itself or by improper timing of position reports in the simulation, one can say that the TeSSLa implementation's output is reasonable.

6.3 Simulation Data

MITSIM can generate datasets for a different number of expressways starting with a minimum of 0.5 expressway which can be increased in steps of 0.5. Other parameters as length of simulation, number of generated cars etc. can also be changed, but we leave them at default. A 3 hours simulation starts with a low event frequency which increases until the end. See Figure 6.2 for the frequencies of a default simulation at scale 1.0.

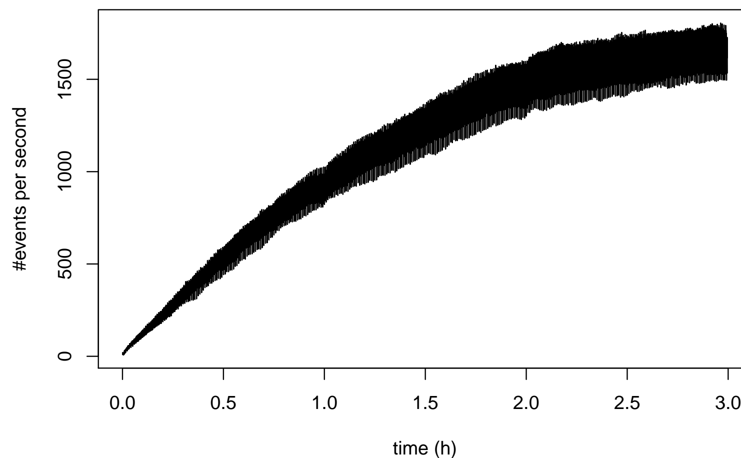


Figure 6.2: Event Frequency over Time for a Default 1 Expressway Simulation

Instead of directly generating a dataset for 0.5 expressways, we use a dataset for a whole expressway and devide it into two 0.5 expressway datasets by splitting the position reports by the direction, dir= 0 for the first and dir= 1 for the second dataset. Additionally we include all query requests from cars which send position reports in the simulation with the corresponding direction. The toll history is taken as it is and is not scaled down which, as we will see, barely affects the performance since it can be loaded before the actual benchmark starts.

For smaller scaled datasets, we will take a 0.5 scaled set and only take cars where the `carid` can be devided by some number, for example 5 to generate a 0.1 scaled dataset. Then we can divide all `carids` by that number, to reduce the maximum `carid`, allowing us to also scale down the toll history. This means that, by applying a scale factor of 0.1 this way, will yield a 0.1 scaled stream data set and a 0.2 scaled history, since the first scale to 0.5 is not applied to the historical data. One thing to notice here, is that each accident produced by MITSIM usually involves two vehicles, therefore scaling down the dataset this way will most likely remove at least one of the accident cars and therefore remove the whole accident. To still keep all accidents as they are, we first detect all accidents in the corresponding direction, and afterwards add reports and query requests of accident cars to the scaled down set again, which have been removed by the scaling.

6.4 Performance Examination

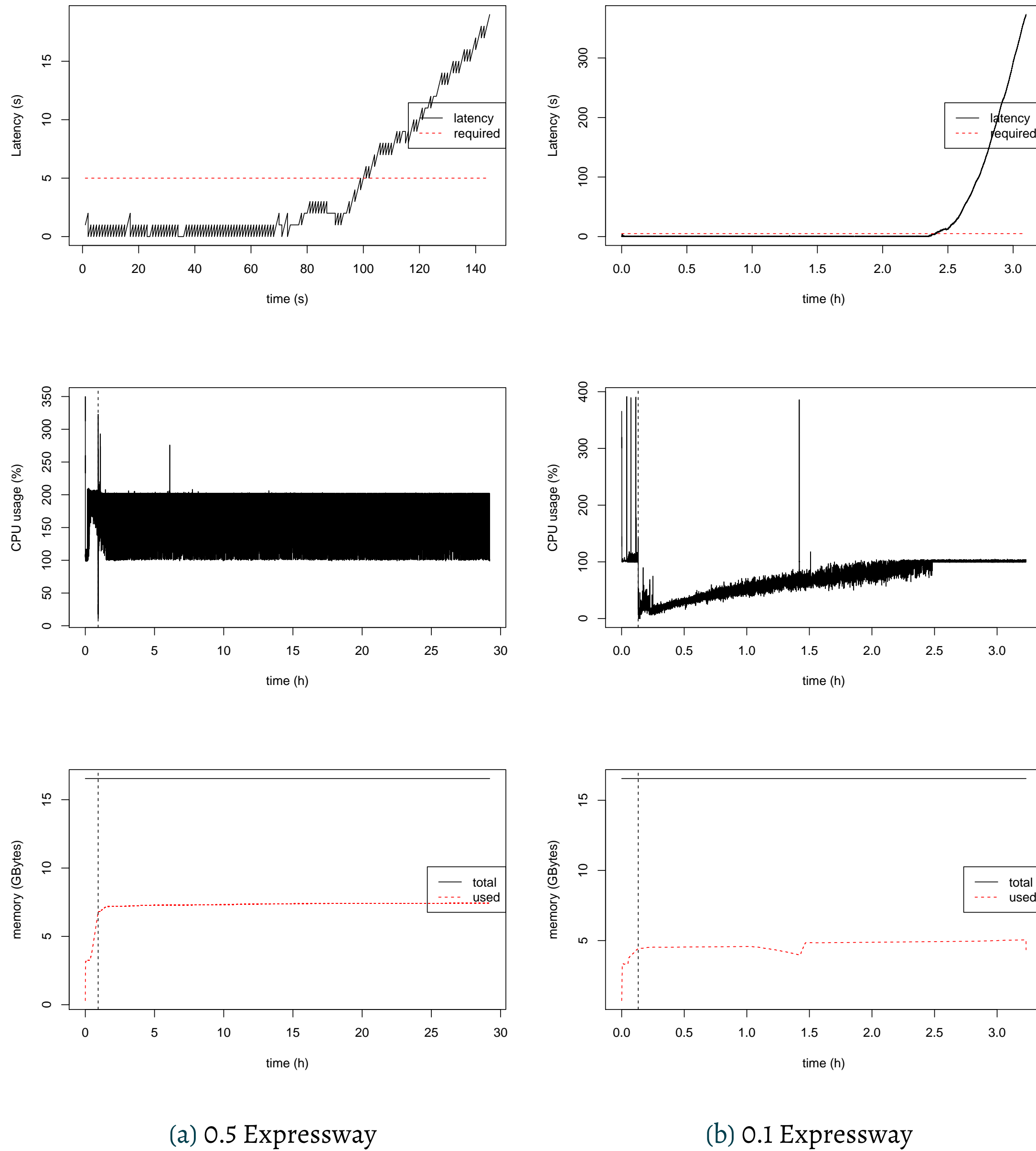
For TeSSLa we will consider two different approaches. First we will focus on the TeSSLa interpereter as it is the most stable implementation offering support for numerous language features. By splitting the specification into two parts to run interpreter instances in a pipeline we still only achieve a comparatively low L-Rating of 0.1. Finally we use the transcompiler [9] to translate the TeSSLa specification to an efficient Scala program.

TeSSLa Interpreter

By running a simulation at the scale of 0.5 expressways on a 2.6 GHz Intel i5, dual-core with 4 threads and 16 GB RAM, the TeSSLa implementation already fails to meet the response time constraints within the first 2 minutes. After that, the response time grows continuously and the whole simulation with three hours of input takes about 28 hours in total to be processed, resulting in a maximal response time of about 25 hours. Figure 6.3 (a) shows the stystem statistics of such a run. Other than the answer latency plots which start with the first second of the simulation, the memory and CPU usage plots start with loading the toll history. The dashed vertical line marks the time where the historical data is completely passed and the simulation starts. The CPU usage plot of the JVM suggests, that the JVM uses background tasks for example for memory organization, while the TeSSLa interpreter fully utilizes a single thread which is not sufficient to keep up with the input stream as the toll alert latency plot indicates, showing the first events with more than 5 seconds latency having been outputted after second 100 from the beginning of the simulation. The memory plot shows the highest increase of used memory at the start and while loading the toll history, whereas the used memory only increases slightly during the simulation. The slight but persistent increase during the simulation could be reduced by deleting the last known position of cars on the exit lane which is not implemented, since the number of distinct cars is limited in a default 3 hour simulation and entries of all cars must persistently be kept for account balances anyway.

For a 0.1 scaled simulation, the system manages to process almost 2.5 hours of input before it falls behind. See Figure 6.3 (b) for the system statistics.

With Figure 6.4 (a), the system statistics for a simulation of 1/12 expressway can be

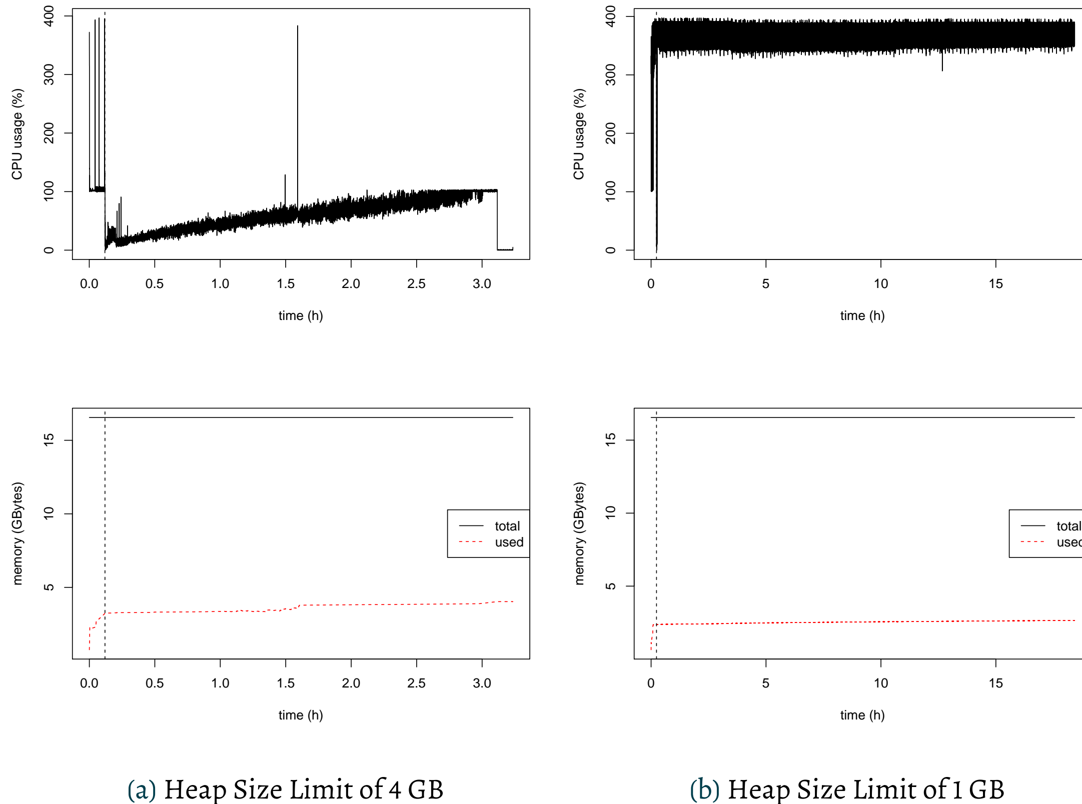


(a) 0.5 Expressway

(b) 0.1 Expressway

Figure 6.3: System Statistics of Simulations where the Response Constraints could not be met. From the top to the bottom they show the latency of outputted toll alerts, the CPU utilization of the JVM and the used memory.

examined where the maximum response time of the systems answer was about 4 seconds, thus the system met all response time constraints. Note that for this run, the heap size of the JVM was limited to 4 GB. The same simulation can also be run with a 1 GB limit but, as can be seen in Figure 6.4 (b), this will leave the JVM busy with memory management, using all of the four available Threads. While the system's answers for such a run where still correct, the three hour simulation of scale 1/12 took the system almost 20 hours.



(a) Heap Size Limit of 4 GB

(b) Heap Size Limit of 1 GB

Figure 6.4: JVM CPU Utilization and Memory Plots for Simulations with 1/12 Expressway.

Performance of Different Specification Parts

The used TeSSLa implementation might not be optimal, therefore we try to identify the most expensive parts of the implementation by running the simulation and only considering one of the four output event types: Focusing the system's capacities on daily expenditure answers only can be implemented the most straightforward way, as it is the least related to the other output events. The implementation still needs to listen to the toll history for initialization, but in the simulation it can ignore position reports and unrelated query requests which makes up more than 99 percent of the stream events. For the accident alerts, with an implementation like presented in Chapter 4, the toll history and all query requests can be ignored. Toll alerts and account balance answers however

can hardly be separated as the main part in both is to compute the tolls. To compute the tolls, the current accidents are required, but since the set of accidents is empty most of the time or does only contain a single entry for low scaled simulations, we will assume that there are no accidents, accepting some toll values falsely to be non-zero. For the current toll values, the segment history over the last five minutes as well as the current minute must be maintained including the number of cars, the average velocity and toll values. It can be implemented using a map with the minute number modulo 6 as key for that minute's segment history, always replacing old entries if a key is repeated. The segment history for a single minute might be implemented by using a (`XWay`; `Seg`; `Dir`) tuple as key for that segment's car statistics. Using this, the toll alerts can be generated; for the account balance we also need to sum up all tolls for each car which have actually been charged, for example by using a map carrying an entry for each car with the charged toll sum and a time value of the last update for query answers. Therefore account balance answers are less frequent than toll alerts, but calculating them requires huge parts of toll alerts plus maintaining the current day's history.

Running the benchmark while solely focusing on single parts shows, that the system is able to handle 0.1 expressways with a maximum response time of 4 seconds in toll alerts and 1 second for all other ouput events. The system statistics are shown in Figure 6.5. These results suggest to implement the benchmark using two TeSSLa interpreter processes, one for accident detection and daily expenditure answers and one for toll alerts and account balances. Since the accidents are needed for correct tolls, the process handling them could directly read the input stream, and pass the position reports combined with accident information to the toll process. This way, the TeSSLa implementation can completely handle the 0.1 scaled simulations while meeting the response time constraints.

Furthermore, the JVM's CPU usage for only handling daily expenditures showed that the system could actually handle a much higher request frequency and indeed, it succeeded to perform on an a scale of 2 expressways with a maximum response time of less than one second. This shows, that the TeSSLa interpreter can handle big sets of stored data and even access it with reasonable frequency as long as the dataset it rarely updated. The system statistics in Figure 6.6 show that there is still room for further CPU utilization. The toll history for this simulation contained 19 million lines, as noticed earlier, about twice as many as for a 0.5 expressway simulation (Figure 6.3 (a)).

TeSSLa-To-Scala Transcompiler

The transcompiler [9] translates the TeSSLa sepcification for Linear Road to a Scala program which operates on the input streams as the interperter run with the specification. It was able to handle a simulation of full 10 expressways. The system statistics plots in Figure 6.7 and Figure 6.8 reveal that the most expensive events are those triggering updates to large datasets: During the initialization the toll history is passed to the system which has to be stored completely. Therefore the implementation will add an entry to its continuously growing datasets for each input event. This would not be a problem for simple inserts on mutable data structures, i.e. structures which are not persistent and can be changed. But since variables in TeSSLa have an immutable semantics in gen-

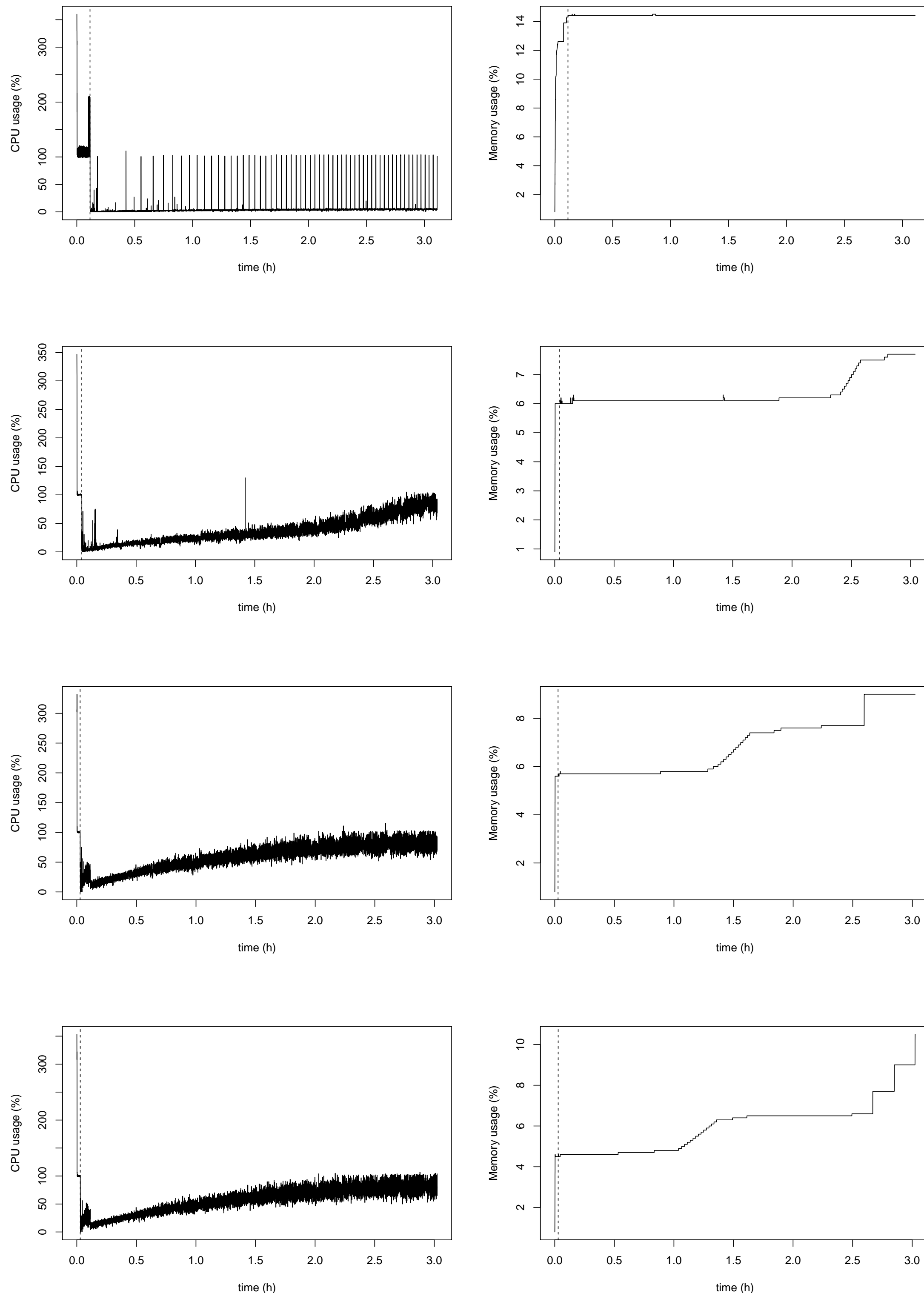


Figure 6.5: JVM CPU and Memory Utilization for 0.1 Expressway Simulation, only handling a single output Event Type. From the Top to the Bottom the Focus was on Daily Expenditures, Accident Alerts, Toll Alerts and Account Balances.

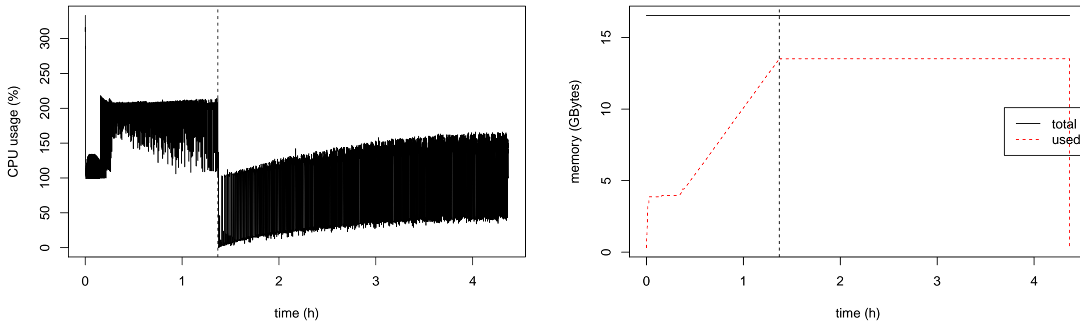


Figure 6.6: JVM Statistics for Daily Expenditures on 2 Expressway Simulation.

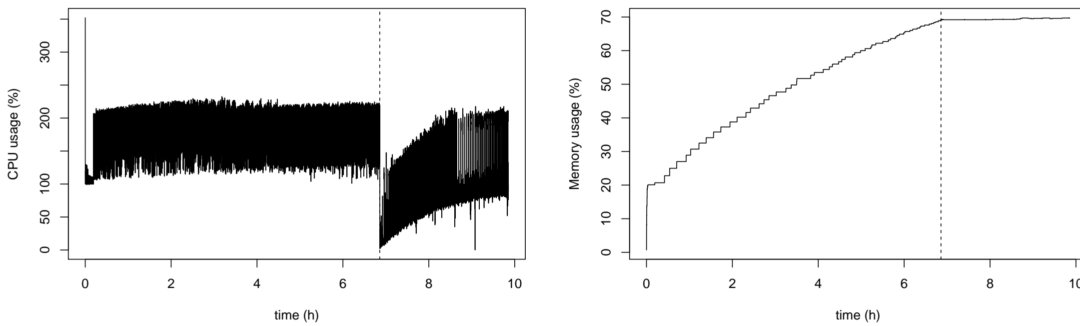


Figure 6.7: JVM Statistics for a 10 Expressway run with the Scala version.

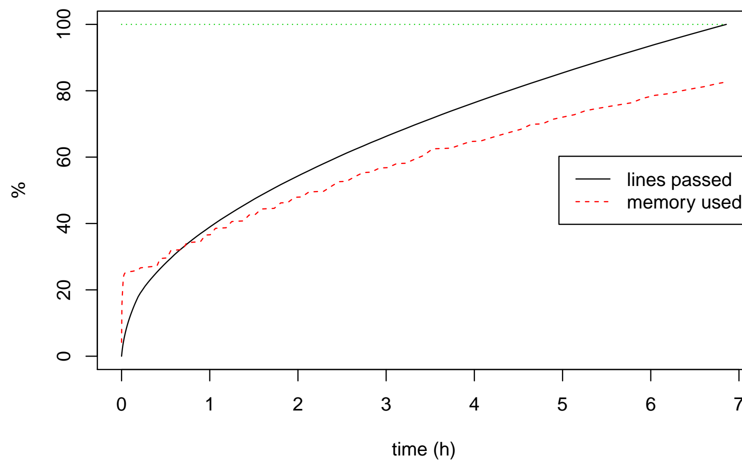


Figure 6.8: Memory Statistics for loading the Toll History of a 10 Expressway simulation with the Scala version.

eral, as described in [9], a straightforward implementation always uses immutable data structures. This requires to copy the whole structure for changes, causing the execution to slow down with increasing dataset sizes. A generated program should therefore use mutable datastructures whenever allowed by the data dependencies from its specification. Since the problem of finding variables which could be implemented with mutable structures has shown to be NP-complete in general, the transcompiler uses an approximation algorithm. In the case of the used TeSSLa specification for Linear Road, the compiler found that all structures must be immutable. For the 10 expressway simulation the toll history contained roughly 95 million entries (~1.4GB input file size) which took the system about 6 hours to store, whereas the actual simulation with about 120 million events (~6.5GB input file size) could be processed within the 3 hours of the benchmark, with a maximum response time of about 2 seconds. Carefully planning the access of data structures which could be implemented mutable, might enable the compiler to detect them and signifficantly improve the performance.

Comparison to other Systems

With the interpreter we could handle simulations up to 0.1 expressways and therefore achieved an L-Rating of 0.1 on a 2.6 GHz Intel Core i5 with 16 GB RAM running Linux. In the original paper of the bechmark [2] they presented two implementations, both running on a 3 GHz Pentium box with 2 GB RAM and a Linux oparating system. The first used a commercially available Relational Database which they called System X and the other utilized a pre-release commercialization of their DSMS Aurora. With System X they achieved an L-Rating of 0.5 and for Aurora a rating of 2.5.

With the MaxStream project [4] they describe an architecture integrating DSMSs together with traditional DBMSs to gain flexibility in the application domain. Their implementation extends SAP MaxDB by a commercially available DSMS using an SQL-based query language which they called SPE X (Stream Processing Engine X). They implemented the benchmark for both, their combined implementation as well as solely using SPE X to observe the overhead their implementation introduces. The client writing the input stream events and SPE X each ran on a 4-way dual-core AMD Opteron 2.2 GHz with 64 GB memory, the MaxStream server machine was a 2-way quad-core Intel Clovertown 1.86 GHz with 16 GB memory. All machines were running Linux. In both approaches, their MaxStream implementation and the plain SPE X, the system first slightly fails to meet the response time constraints for 5 expressways therefore their run with the highest workload still meeting the constraints indicates an L-Rating of 4.

An implementation with Streamonas DSMS could handle 10 expressways with an average query latency of 26 microseconds [11]. Using the Scala version, we showed that an implementation with TeSSLa can, while meeting all latency constraints, also handle simulations for full 10 expressways, which might be seen as the maximum number as it is implied by the dimensions of Linear City given in [2]. In theory one might further scale the simulations using many more expressways to show more accurate results on the L-Rating, but up to this point we have already seen, that the L-Rating for TeSSLa using a compiled Scala version is at least 10 which can be said to be efficient in comparison to other known results. Furthermore, for simulations of higher scale, one might con-

sider to improve the implementations for the simulation data generator as well as the validator, since on the used system they required significantly more time than the actual benchmark. Therefore the TeSSLa Interpreter implementation shows comparatively low performance on this benchmark, the bottleneck can be seen in frequent changes to massive datasets, but with a transcompiled version, TeSSLa can be used for efficient stream proccessing, as shown with this typical use case for general purpose DSMSs.

7

Conclusion

We have seen that we can use CQL, PipelineDB's SQL and TeSSLa for all of the shown streaming applications, which is not surprising after expressiveness results for these languages, but we further explored how easy certain features can be implemented, providing a rough overview on which of these languages might be the best fit for different use cases: First of all there might be no DSMS fully supporting CQL until now, which disqualifies it for practical use. However it shows different window functions and relation to stream operators as a general idea how relations can be a handy concept for streaming applications, which could be used to lead further research.

PipelineDB as an extension of PostgreSQL features rich and widely used operations on relations as well as multiple tools for turning streams into relations and special stream to stream operators; in general, making data stream processing accessible to a broad user group. But when it comes to more complex ordering and timing patterns which can hardly be expressed using common declarative SQL methods, it demands a detailed understanding of PipelineDB's features.

TeSSLa is comparatively easy to learn because of its small basic operator set and very well suited for ordering and timing patterns which can be implemented using memory of fixed size. In these cases recursive use of basic operators can be utilized to build domain-specific libraries allowing to write easy to use and well readable specifications, even for complex use cases. When it comes to applications requiring to save and diversely access potentially unbounded amounts of data, TeSSLa becomes less agile due to its limited unbounded data structures.

The SRV implementation utilizing PipelineDB exemplarily shows suitability of other DSMSs for SRV, particularly DSMSs with relational approach, but it remains open, how they perform compared to TeSSLa in typical SRV scenarios, especially when TeSSLa allows to make use of hardware acceleration. For TeSSLa's performance on typical use cases for general DSMSs, the Linear Road implementation showed, that the TeSSLa interpreter's performance is inferior compared to other DSMSs for specifications making heavy use of unbounded data structures while the transcompiler yields an efficient solution.

Bibliography

[1] Arasu, A., Babu, S., and Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In: *VLDB J.* 2, Mar. 2004. DOI: 10.1007/s00778-004-0147-z.

[2] Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., and Tibbetts, R. Linear Road: A Stream Data Management Benchmark. In: Oct. 2004. DOI: 10.1016/B978-012088469-8/50044-9.

[3] Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J. STREAM: The Stanford Data Stream Management System. In: Jan. 2006.

[4] Botan, I., Cho, Y., Derakhshan, R., Dindar, N., Haas, L. M., Kim, K., Lee, C., Mundada, G., Shan, M.-C., Tatbul, N., et al. *Design and Implementation of the MaxStream Federated StreamProcessing Architecture*. en. Tech. rep. Zurich, 2009.

[5] Convent, L., Hungerecker, S., Scheffel, T., Schmitz, M., Thoma, D., and Weiss, A. Hardware-Based Runtime Verification with Embedded Tracing Units and Stream Processing: 18th International Conference, RV 2018, Limassol, Cyprus, November 10–13, 2018, Proceedings. In: Nov. 2018, pp. 43–63. ISBN: 978-3-030-03768-0. DOI: 10.1007/978-3-030-03769-7_5.

[6] Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M., and Thoma, D. TeSSLa: Temporal Stream-based Specification Language. In: Aug. 2018.

[7] *Finding a Timing Bug*. 2018. URL: http://training.coems.eu/tutorials/timing/ (visited on 02/22/2020).

[8] Hanif, M., Yoon, H., and Lee, C. Benchmarking Tool for Modern Distributed Stream Processing Engines. In: *2019 International Conference on Information Networking (ICOIN)*. Jan. 2019, pp. 393–395. DOI: 10.1109/ICOIN.2019.8718106.

[9] Kallwies, H. Efficient Code Generation for Stream-based Specifications. In: May 2019.

[10] Law, Y.-N., Wang, H., and Zaniolo, C. Query Languages and Data Models for Database Sequences and Data Streams. In: Dec. 2004, pp. 492–503. DOI: 10.1016/B978-012088469-8/50045-0.

[11] Michael, P. and Parker, D. Real-time spatio-temporal data mining with the "streamonas" data stream management system. In: May 2009, pp. 113–122. ISBN: 9781845641849. DOI: 10.2495/DATA090121.

[12] Mueen, A., Keogh, E., Zhu, Q., Cash, S., and Westover, M. B. Exact Discovery of Time Series Motifs. In: vol. 2009. Apr. 2009, pp. 473–484. DOI: 10.1137/1.9781611972795.41.

[13] *PipelineDB. High-performance time-series aggregation for PostgreSQL*. 2019. URL: https://www.pipelinedb.com/ (visited on 01/04/2020).

[14] *PipelineDB Docs. Documentation*. 2019. URL: http://docs.pipelinedb.com (visited on 01/04/2020).

[15] *PipelineDB Use Cases*. 2018. URL: https://www.pipelinedb.com/use-cases (visited on 02/26/2020).

[16] *PostgreSQL. The World's Most Advanced Open Source Relational Database*. 2019. URL: https://www.postgresql.org/ (visited on 01/04/2020).

[17] *Processing time series data: What are the options? Get your data from everywhere you can, anytime you can, they said, so you did. Now, you have a series of data points through time (a time series) in your hands, and you don't know what to do with it? Worry not, because there's a bunch of options.* Sept. 28, 2018. URL: https://www.zdnet.com/article/processing-time-series-data-what-are-the-options/ (visited on 02/13/2020).

[18] *Runtime Verification With TeSSLa*. 2018. URL: https://www.tessla.io/rv-tutorial/ (visited on 03/22/2020).

[19] *Stream Query Repository*. Dec. 2, 2002. URL: http://infolab.stanford.edu/stream/sqr/ (visited on 01/04/2020).

[20] *TeSSLa: Temporal Stream-based Specification Language*. 2018. URL: https://www.tessla.io/ (visited on 02/25/2020).

[21] *TeSSLa web IDE*. URL: https://tessla.isp.uni-luebeck.de/ (visited on 02/25/2020).

[22] *The State of the Time Series Database Market*. Apr. 3, 2018. URL: https://redmonk.com/rstephens/2018/04/03/the-state-of-the-time-series-database-market/ (visited on 02/13/2020).

[23] *Why time series databases are exploding in popularity. Time series databases are on the rise, with TimescaleDB of particular interest to developers.* June 26, 2019. URL: https://www.techrepublic.com/article/why-time-series-databases-are-exploding-in-popularity/ (visited on 02/13/2020).

[24] Yang, Q. and Koutsopoulos, H. N. A Microscopic Traffic Simulator for evaluation of dynamic traffic management systems. In: *Transportation Research Part C: Emerging Technologies* 4(3):113 –129, 1996. ISSN: 0968-090X. DOI: https://doi.org/10.1016/S0968-090X(96)00006-X. URL: http://www.sciencedirect.com/science/article/pii/S0968090X9600006X.

[25] Zhang, Y., Pham Minh, D., Corcho, O., and Calbimonte, J.-P. SRBench: A streaming RDF/SPARQL benchmark. In: vol. 7649. Nov. 2012, pp. 641–657. DOI: 10.1007/978-3-642-35176-1_40.