UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Data Science Curriculum

Agnes Annilo

Exploring SQL-based near real-time conformance checking using Stream Processing Engines

Master's Thesis (15 ECTS)

Supervisor(s): Kristo Raun, MSc

Tartu 2024

Exploring SQL-based near-realtime conformance checking using Stream Processing Engines

Abstract: Conformance checking is used to validate the accurate completion of business processes against an existing process model. Effective conformance checking using event logs has been a vital strategy in ensuring the operational integrity and compliance of a business. With the development of streaming platforms, which offer near real-time data processing, non-conformance can be identified quickly and accurately. Streaming engines like Kafka or Spark are designed to process data in near real-time, in addition they are optimised for SQL. This thesis explores the possible application of conformance checking using SQL with Kafka and ksql or Spark Structured Streaming in the conformance checking of near real-time data streams. The thesis presents a structured investigation into stateful processing of events using a combination of Spark Structured Streaming and Spark SQL, alongside Kafka and ksqlDB. The challenges of handling event streams sequentially as well as implementing a stateful solution are discussed.

Keywords: Process mining, conformance checking, SQL, Spark, Kafka, ksqlDB

CERCS: P170 Computer Science, Numerical Analysis, Systems, Control

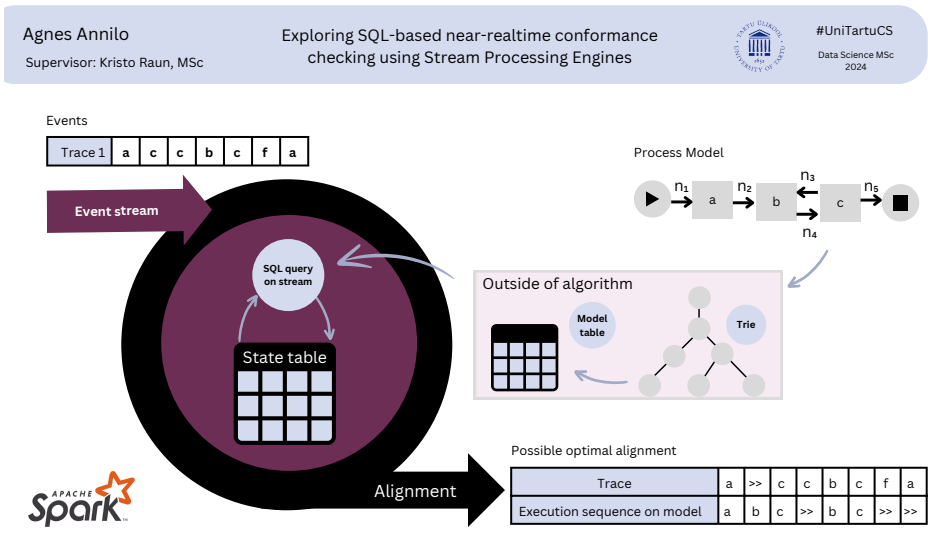


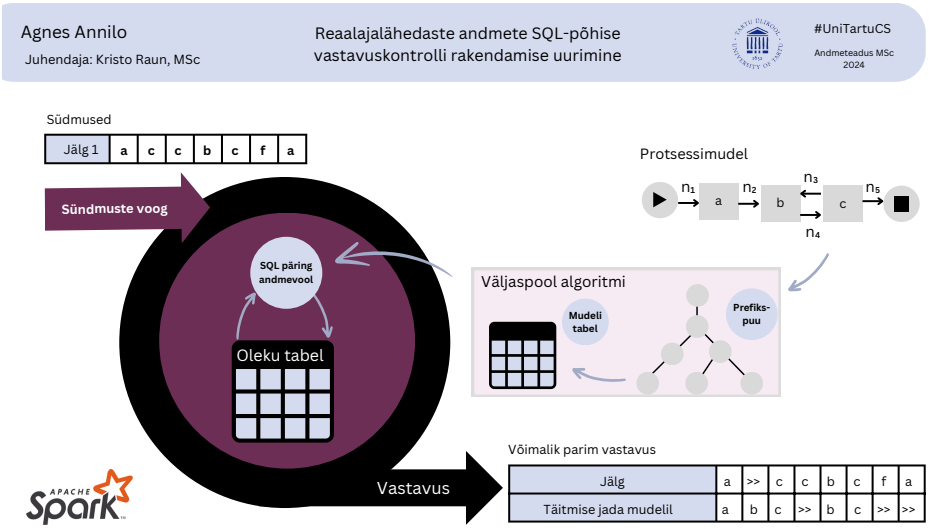
Figure 1. Graphical abstract

Reaalajalähedaste andmete SQL-põhise vastavuskontrolli rakendamise uurimine

Lühikokkuvõte: Vastavuskontrolli kasutatakse äriprotsesside korrektse toimimise valideerimiseks. Sündmuste logide vastavuse kontrollimist kasutatakse tihti terviklikkuse ja regulatsioonidele vastavuse tagamisel. Andmevoogude töötlemisele keskenduvate platformide arenguga, mis pakuvad peaaegu reaalajas andmetöötlust, saab mittevastavust kiiresti ja täpselt tuvastada. Andmevoogude töötlemise platformid nagu Kafka või Spark on kujundatud andmete töötlemiseks peaaegu reaalajas ja on optimeeritud SQL-i jaoks. Magistritöös uuritakse vastavuse kontrolli võimalikku rakendamist, kasutades SQL-i koos Kafka ja ksqlDB või Spark Structured Streaming platformiga reaalajas andmevoogude vastavuse kontrollimisel. Magistritöös vormistatakse nende platformide kasutamise võimalused ning puudused.

Võtmesõnad: Protsessikaeve, vastavuskontroll, SQL, Spark, Kafka, ksqlDB

CERCS: P170 Arvutiteadus, Arvutusmeetodid, Süsteemid, Juhtimine



Joonis 2. Graafiline abstrakt

Contents

1 Introduction

Process mining is a method used to improve business operations by leveraging event data from IT systems. These events are stored in event logs, which are used to identify and anticipate performance and compliance issues. One of many techniques used to improve performance and compliance is conformance checking. Conformance checking requires a process model, which describes how a process should work. Incoming event log data is used to monitor deviations from the expected process flow described by the model. Traditionally, conformance checking is done offline. However, for businesses that require near real-time feedback on the correctness of their processes, such as banks or manufacturers, conformance checking can be done online (in near real-time). Only recently has online conformance checking been more thoroughly researched, and only some approaches have been explicitly designed for online conformance checking [1, 2]. A solution based on SQL was preferred in this thesis, as SQL-based systems often offer better interoperability with various data sources and other SQL-based systems, reducing the complexity and overhead associated with data integration and migration tasks. As SQL is standardised, similar solutions can be used regardless of the underlying platform.

The standard way to monitor whether an event is conformant is by using alignments [3]. For streaming contexts, prefix alignments are used instead of complete alignments, as we can not conclude that any given event completes the trace. Currently, streaming conformance checking methods that produce alignments as output are not written using SQL; instead, they are written in Python or Java [4, 5, 6].

The objective of this thesis was to to investigate possible solutions for a streaming data conformance pipeline using SQL-based solutions. Kafka with ksqlDB and Spark Structured Streaming, both data platforms specifically designed for streaming. As every incoming event must be able to reference the already existing prefix alignments, a stateful solution is required. Spark supports handling states in Scala and Java. However, more challenging queries in SQL involving states have yet to be explored. The solution is inspired by an already existing state-of-the-art method, the IWS algorithm [4], but has been restructured to be implemented on a streaming platform that is scalable, fault-tolerant, and widely used in the industry.

The process model is created using a Python script, after which it is converted to an SQL table with additional transformations, imitating a trie (prefix tree) structure. The event data can be streamed from an existing log or other streaming connectors and used in a streaming query to calculate its fit in the model based on the prefix alignments. This thesis investigates the possible solutions to stateful processing of events using Spark Structured Streaming and Spark SQL, in addition with Kafka and ksqlDB. The potential shortcomings of handling sequential processing using these solutions are discussed.

The thesis will outline the key concepts in process mining and conformance checking in section 2 and provide a background on Apache Spark and other tools used in section 3. In section 4, the problem is explained in detail, and the shortcomings of using Spark to

solve this issue are discussed. The different approaches developed and implementations will be presented and discussed in section 5, and in section 6, the evaluation of these methods and further additions are reported. Section 7 concludes the thesis, providing an overview of the problem, limitations and solutions.

The typing assistant Grammarly [7] was used in this thesis to improve formatting.

2 Background

The Cambridge Dictionary [8] describes a process as a series of actions to achieve a result. Businesses are built on a collection of interconnected processes with distinct objectives. Structures have been established to enhance the effectiveness of the processes, improving management and business decisions. These decisions include optimising cost, discovering system bottlenecks, and understanding common user actions. As a business grows, so does the complexity of its business processes. Thus, it is only logical that these processes are improved, optimised and automated to make better business decisions based on data recorded about these processes. Not only has process mining helped with lowering business costs, but it can also help companies be more sustainable to reduce energy consumption [9].

2.1 Process mining

The theory of process mining described in the thesis is predominantly based on the research of Wil van der Aalst since the late 1990s. Van der Aalst, the pioneer of contemporary process mining, has significantly influenced this field [10]. Over the past decade, technological advancements have facilitated the adoption and continuous refinement of process mining methodologies. Process mining [11] is somewhere between data science and process modelling. It is generally centred around analysing logs containing unique identifying information about events. Process mining primarily targets processes characterised by frequently occurring activities collected in event logs. The regular occurrence of these events enables the development of process models that reflect historical patterns and behaviours. Process mining is used to discover how processes work and give factual feedback on business processes. Other business analysis methods focus excessively on data analysis or visualisation without considering the actual processes in an organisation. Processes are investigated in detail using process models, which describe how events or user actions depend on each other [10].

Process mining is divided into three core areas: process discovery, conformance checking and process enhancement [11]. The scope of the thesis is confined to conformance checking; thus, the other areas of process mining will not be explored.

2.2 Process model

The following subsection is referenced from the chapter in the textbook [12] by Alejandro Vaisman.

A process model is defined as a formalised view of a business process, represented through a series of connected activities, either parallel or sequential, aimed at achieving a specific goal. In other words, process models describe common process patterns, often represented graphically to be easier to comprehend. Process models can be visually

depicted in many ways; however, the Petri net is the most commonly used depiction in process mining workbooks. Petri nets are used because they are standardised and formalised and allow concurrency and loops. A Petri net consists of places, transitions and directed arcs; places represent states, transitions represent events, and arcs denote the flow between places and transitions. Petri net transitions can be labelled and can bear the same label. Some transitions are not observable; they are called silent or invisible.

A subclass of Petri nets, known as WorkFlow nets (Figure 3), are often used in business process modelling [11].Workflow nets are labelled Petri nets with a dedicated process start (source) and end place (sink). All nodes are on a path from the source to the sink. A sequence of labelled transitions can be observed when the net is enacted. The sequence of executed transitions is called an execution sequence. The sequence starts from the source and ends at the sink.

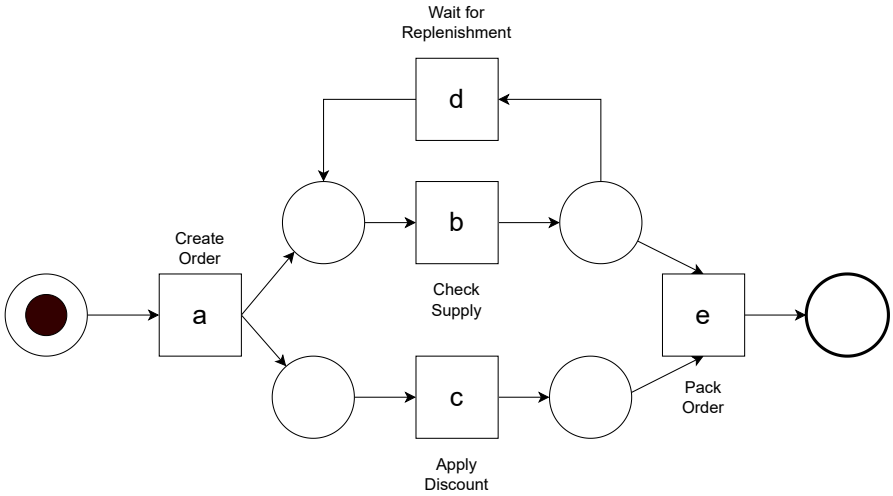


Figure 3. Running example: process model [4]

Workflow nets can model cyclical activities. However, in this thesis, the trie structure representing the model is created based on a finite proxy log (Table 1, which mimicks the proxy behaviour of a model and confines the number of loops to a fixed number.

For this thesis, the process model is sampled from a proxy log but can also be created using process mining.

2.3 Event log and stream

Within the scope of process mining, an event log is a compilation of time-stamped event records created during the operation of a business process. As these logs are what business process models are built upon, the logs must be of high quality. Missing entries,

Table 1. Running example: proxy log.

Case id	Trace
1	<a, b, c, d, b, e>
2	<a, b, c, e>
3	<a, b, c, e>
4	<a, b, d, b, e>
5	<a, b, d, c, b, e>
6	<a, b, e>
7	<a, c, b, d, b, e>
8	<a, c, b, e>

duplicates or faulty timestamps can lead to an unstructured model, which does not reflect users' actual behaviour in the business process [13].

Process mining algorithms operate on an event log, a collection of sequentially recorded events relating to a particular business process instance, also known as a case [14]. Per every case, a trace of a sequence of events is recorded. However, events with the same activity can occur multiple times per trace. In order to distinguish these events, additional metadata, such as the timestamp, is also recorded. The complete set of all traces in the event log represents the entire or partial history of the business process.

An event stream relies on the structure of an event log, as every event must be identifiable and have a timestamp, as well as an activity name or label as seen in Figure 4. However, an event stream can have an infinite sequence of events [15]. Events in event streams are ordered and can be ordered in multiple ways. For this thesis, they are ordered based on their timestamp. Solutions based on event streams are usually built so that every incoming event is processed in near-real time instead of collecting the streams into large files to be processed together.

2.4 Conformance checking

The following subsection is based on the works by W. van der Aalst and J. Carmona and S. van Zelst et al [11, 16, 5]. With the shift towards online transactions, particularly in the past few years, the need for validation that business processes are completed accurately has increased. Large and small firms risk losing significant clients or revenue due to transactional errors or non-compliance due to system glitches. Primarily, these deviations are identified by strict rules before anything happens, yet there remains a possibility of human error or edge cases. Organisations implement protective measures for business-critical transactions. A bank with many transactions or a hospital with hazardous medical procedures involving radiation will have safeguards based on regulations. With the help of conformance checking, it is possible to validate that these safeguards are followed.

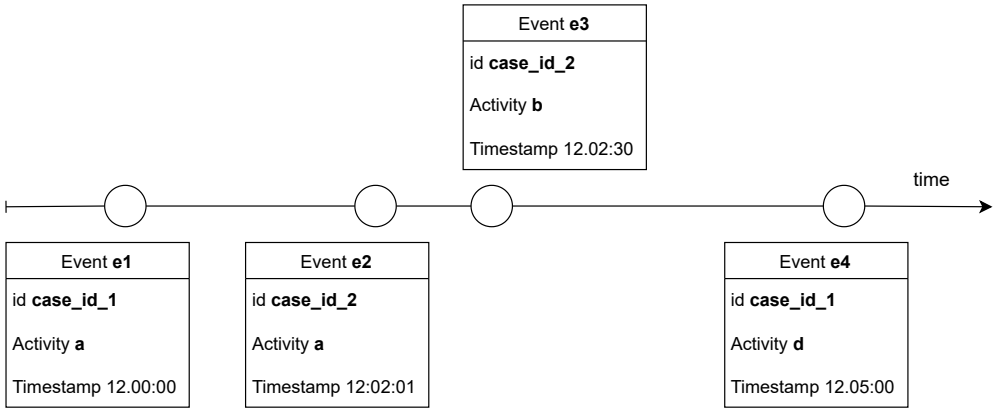


Figure 4. Event stream

Conformance checking finds discrepancies between the expected and actual behaviour of a process by evaluating whether the actual execution of a process aligns with the predefined model (observed vs modelled behaviour). Conformance checking can be used to detect and explain deviations, but also measure the fitness using a cost function. Conformance checking is used to manage the extensive nature of logs, which exceeds the capacity of human monitoring.

The interpretation on what counts as non-conformance is up to the business. There are two ways to describe a model: descriptive and normative. If the model is intended to be descriptive, then discrepancies between the model and logs reflect that the model must be improved in order to capture the reality more accurately. If the model is intended to be normative, then there can be either undesirable or desirable deviations. Conformance checking should take into account both deviations: whether the model needs improvement or the business process truly has many deviations from the model.

There are various techniques for conformance checking, but the two most commonly used are token-based replay and alignments [17]. Token-based alignment replays traces in the event log against the model transitions. This is quite efficient, but valid paths through the process model are not produced. Alignments on the other hand are more detailed and give a better overview of what and at what moment went wrong. As alignments are event-based, it is possible to pinpoint what events are causing non-conformance. Alignments try to find the optimal path through the process model, which is as close to the observed behaviour as possible. The thesis is centred around alignment-based conformance checking, for which two inputs are required: a process model and an event log.

Process mining is mainly done off-line, but some businesses might need to be notified as soon as some non-conformant operation takes place, which is why streaming solutions are becoming more researched.

2.4.1 Alignnments

Most state-of-the-art approaches for conformance checking rely on the concept of alignments of the event log and process model [18]. An alignment denoted by γ directly connects an observed trace σ in the event log to an execution sequence π of the model most similar to σ. The next section is based on the article by J. Carmona et al [17].

An alignment is represented by a two-row matrix, where the first row consists of activities in the trace and a special symbol $>>$, which represents skips in the trace. The second row describes the transitions in an execution sequence of the process model and a special symbol , which represents skips in the execution sequence. A column with both values being skip symbols $(>>, >>)$ is illegal. For a complete alignment, the sequence of activities ignoring skips yields the trace, and the sequence of transitions yields the complete run of the net. Each event in the trace and transition in the model, which creates a column in the matrix, is paired as a tuple (e_i, a_i) and called a move of the alignment. The thesis considers three types of moves:

Synchronous move: The activity of the trace e_i and the model transition a_i correspond to each other, so $a_i = e_i$. This is the expected situation as the model corresponds to the event activity, e_i and $a_i \neq >>$.

Model move: When the model expects an activity from the trace, however, there is no related activity in the trace. This move represents a deviation between the trace and event model, where we will execute a transition on the event model, but that transition is not present in the trace, so that event is skipped. So $e_i =>>$ and $a_i \neq >>$.

Log move: The counterpart of a model move, where the trace contains an activity which should have not been executed based on the model. This describes an unnecessary activity in the trace, meaning $e_i \neq >>$ and $a_i =>>$.

Table 2. Example of a complete alignment

σ	a	b	b	d	$\gg$	c	b	$\gg$
π	a	b	$\gg$	d	b	c	b	e

Model and log moves are referred to as asynchronous moves. Alignments are constructed of these three moves. Alignments which have the least amount of log or model moves are called optimal alignments. Optimal alignments are mostly preferred as they align the events of a particular case to the closest matching path permitted by the process model [18]. It is also possible to separate the deviations from the model based on a cost function, assigning either the same or different costs to log and model moves. The most common cost function assigns the cost of model and log moves as 1 and synchronous moves as 0. For this cost function, based on the example alignment in Table 2, the cost is equal to the number of skip activities - 3.

In a streaming setting the alignment is computed based on the events that have occurred at the point of examination and thus the complete alignment is not computed,

as the algorithm can expect further events. The resulting alignment is called a prefix alignment. In Table 3, the prefix alignment is separated from the complete alignment, which illustrates an alignment in a streaming context, where it is possible that an event with the activity e might still occur in the future and so it is not considered in the current state of the alignment and is not seen as a move in the alignment.

Table 3. Example of a prefix alignment. The gray event e can still occur.

σ	a	b	b	d	$\gg$	c	b	$\gg$
π	a	b	$\gg$	d	b	c	b	e

2.4.2 Trie

A trie [19] also known as a prefix tree is a particular type of tree, where all the children of a node have a common prefix, illustrated in Figure 5. The basis of the thesis was to create a similar solution to the already-existing algorithm I Will Survive, which stores the model as a trie. For the thesis, the trie is generated from an existing event log, using the same sample data described in the IWS article and stored in a database table. The table containing the trie structure is referenced as the model table, as no actual process model is created. If we consider our process model to be sequential with no infinite looping, storing the model this way is rational, as it naturally aligns with the sequential progression of events. The suitability of an event within the model is evaluated by examining the compatibility between the existing prefix in the trie and the sequence of incoming events. The trie is computed during pre-processing offline and remains unaltered during the execution of the conformance checking stream.

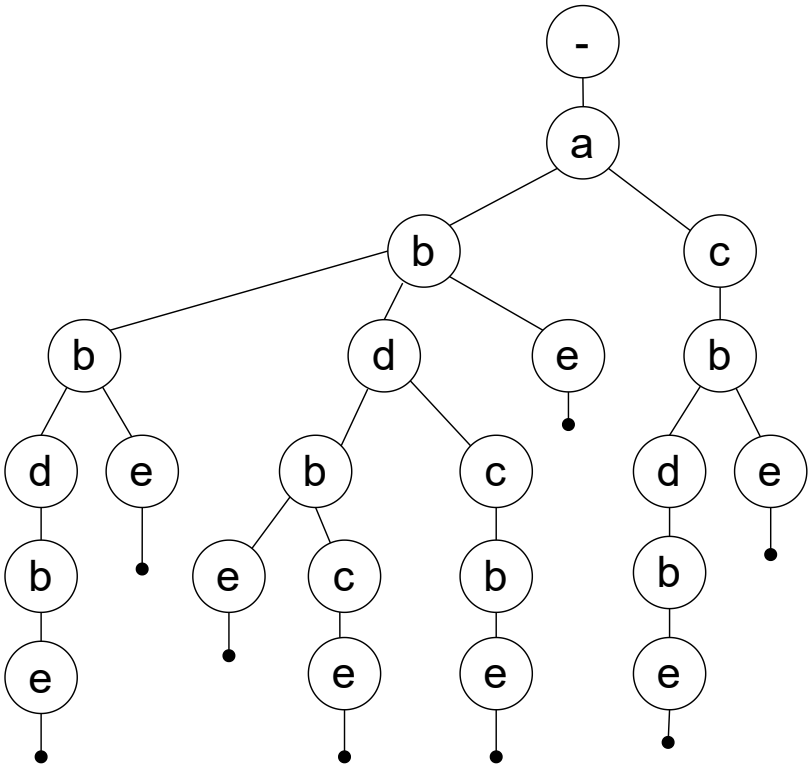


Figure 5. Running example: trie [4]

2.4.3 XES file format

Typical log storing formats are eXtensible Event Stream (XES) and Mining eXtensible Markup Language MXML, both being XML files. Created in 2003, the MXML used to be the standard of log storing [11]. However, using MXML has its limitations, as it is not expandable and is restrictive in using new data types.The solution for this problem was to create a tag-based format called XES [20]. XES has been the process mining standard since 2010, when it was adopted by the IEEE Task Force on Process Mining [21]. An XES file contains any number of log traces, each trace has any number of events corresponding to a particular case. [11]

In the structure of an XES log, event data begins with a $< log >$ tag, followed by $< trace >$ tags, indicating traces, which are identified by a unique case ID stored in the concept:name attribute. Within each trace, individual events are captured between $< event >$ tags. Events are primarily characterised by the *concept:name* and *time:timestamp* attributes, specifying the event's activity name and event time. Events may also contain additional attributes, however the minimal attributes for constructing

an event log include case ID, event activity name and timestamp.

3 Used tools

The final solution was implemented using Apache Spark: PySpark, Spark SQL and a Python library called PM4PY - Process Mining for Python [22]. PM4PY was used to read the XES log files into a Pandas DataFrame, then converted to the required structure using Python. Additionally a solution based on Apache Kafka and ksql was tested as well.

3.1 Apache Kafka and ksql

Kafka, a distributed system for stream processes, utilises the publish/subscribe model, where the messages are stored and sorted in defined topics. In Kafka, producers send messages to brokers distributed across a cluster's multiple nodes. KsqlDB is a streaming database designed to enable real-time data processing and analytics. The domain-specific language used in ksqlDB is ksql. Streams can be created from existing Kafka topics or streams using create stream as (CSAS) queries. Tables can be created from streams using create table as queries (CTAS) [23].

3.2 Alignnments

The following chapter on Spark architecture is based on the Apache Spark documentation [24].

Apache Spark, developed at UC Berkeley's AMP lab, is an open-source cluster-computing framework that supports streaming and batch data processing. Spark is preferred due to its in-memory cluster computing, where the disk is only interacted with during data loading and the results are saved to memory. Different levels of caching are supported, but Spark only uses the storage level of memory by default, which is the storage of deserialised objects in memory. The hard drive is used if data does not fit into the memory.

Apache Spark is a widely used framework due to its many positive features, such as speed, usability, advanced analytics, multi-platform support, in-memory computing, and near real-time stream processing. It is specifically designed for computational tasks that involve reusing data across multiple parallel operations [25].

In Spark, the architecture is based on a master-worker system, where the driver program functions as the master node and the executors operate as worker nodes as seen in Figure 6. When a job is submitted to Apache Spark, the driver program initiates the *SparkContext*, a class responsible for accessing Spark functionality, which coordinates all activities within the cluster. Upon receiving an input file, executors start processing and

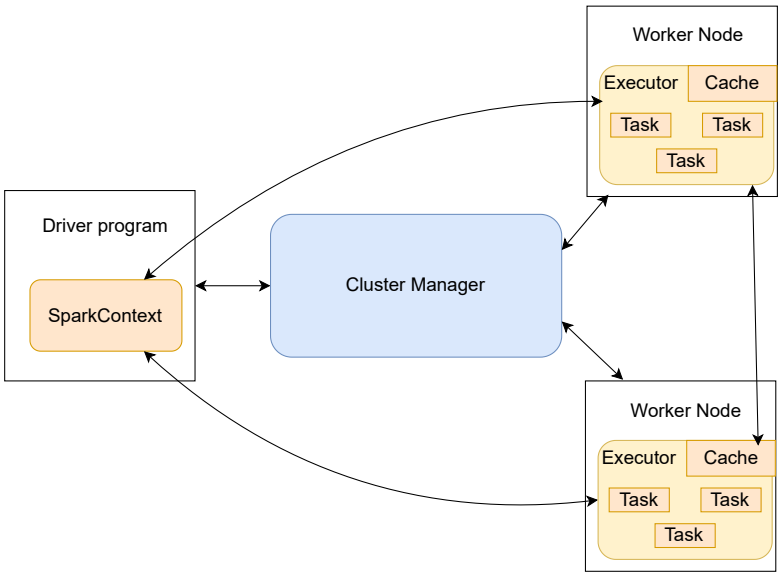


Figure 6. Components of a spark cluster [26].

maintain an active status throughout the job, utilising multiple CPU threads to execute tasks in parallel [27].

The basis of the in-memory computation are Resilient Distributed Datasets (RDDs) and Directed Acyclic Graphs (DAGs). Any job submitted in Spark creates a DAG, which divides the job into different stages. Each stage is made up of tasks determined by the input data and RDD partitioning.

An RDD is immutable and forms a distributed collection of objects segmented into logical partitions processed on different cluster nodes. RDDs are fault-tolerant, storing data in multiple locations, and use lineage graphs. RDDs support parallel operations, allowing simultaneous processing across multiple nodes, but are manipulated as a single logical entity. Spark manages the distribution of computations to all partitions within the RDD and oversees the necessary data redistributions or aggregations. RDDs support two primary operations: transformations and actions. Transformations generate new datasets from existing data, while actions compute results from datasets, returning values to the driver program. In Spark the Core API manipulating RDDs directly is considered a low level API and is less optimised. Therefore, queries are typically structured using high level interfaces like DataFrames or Datasets that are built upon RDDs, but are much more efficient [28].

Datasets are distributed collections of data, with additional information about data structure and computations. The Dataset API is available in Scala and Java only. DataFrames are Datasets organized into named columns, similar to relational database tables. DataFrames are available in Python as well. When using the SQL API within

another programming language, the results will be returned as a Dataset or DataFrame.

Spark is implemented in Scala, but supports high-level APIs in Java, Python and R in addition to Scala. Spark has a module for structured data processing using SQL called Spark SQL. Spark uses the same execution engine independent of API or language, allowing developers to switch between APIs easily. The scope of this thesis was to examine conformance checking streaming solutions using Spark SQL and the Pyspark API.

3.3 Spark SQL

Spark SQL is a module within Apache Spark designed for processing structured data through an abstraction called DataFrame, which extends the capabilities of RDDs by adding schema information. The addition of schema information and computation knowledge allow Spark SQL to perform extra optimizations. A DataFrame, organised into named columns, is analogous to a relational database table or a Pandas DataFrame, yet it is optimised more effectively for distributed processing [29].

Spark SQL allows a user to register any DataFrame as a table or view, which can then be queried using standard SQL. Both DataFrames and SQL share the same optimization and execution pipeline. Figure 7 shows the Spark SQL programming interface.

3.4 Spark Structured Streaming

The following subsections are based on the Structured Streaming Programming Guide by Spark [31].

Structured Streaming is a high-level API for stream processing built on the Spark SQL engine. It allows users to define streaming computations with the same syntax used for batch processing on static data using the Dataset/DataFrame API (DataFrame for PySpark). This model simplifies streaming by handling incremental, continuous computation and updating results as new data arrives. The platform supports end-to-end exactly-once processing through checkpointing and Write-Ahead Logs.

Structured Streaming operates on a micro-batch processing model (Figure 8), processing streams in small batches; however, a continuous processing mode, although experimental, is available as well. The Spark streaming engine periodically monitors the data source and executes a batch query on the new data captured since the last batch execution.

The driver program saves the offsets of records it will process into the Write-Ahead Log, hence in case there is a failure and the streaming job needs to be restarted, Spark can resume processing from the exact failure point. The range of offsets to be processed in the next micro-batch is logged, ensuring that each batch can be re-executed deterministically [32]. A given batch will always consist of the same data and a completed batch of events is recorded in the commit log with it's batch id [33].

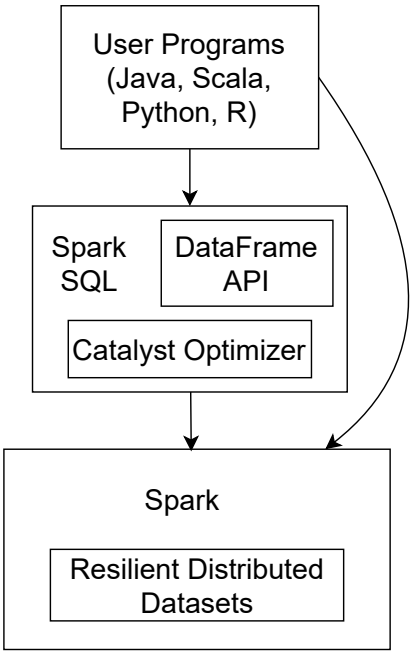


Figure 7. Spark SQL programming interface [30].

Spark Structured Streaming processes a stream of data as a dynamic table, that expands continually with new entries. Outputs from these queries populate an intermediate *Result Table*, either by appending or updating the table to reflect the changes, written to an external sink after processing has concluded. The frequency of data processing can be controlled by setting a trigger interval, however by default a new run is initiated as soon as the previous one has concluded.

Structured Streaming is distinct from other streaming platforms as it does not materialise the entire input table. Instead it processes incoming data incrementally, updating the result based on the latest data while discarding the original input after processing. Due to this approach, only necessary intermediate results, such as counts, are stored with the other state data in the *Result Table*.

3.4.1 Input modes

A streaming DataFrame can be created using the *DataStreamReader* interface. The preferred stream sources are either a file source or a Kafka source, other sources are usually used for testing. Tables can also be used as inputs and as sinks using Streaming

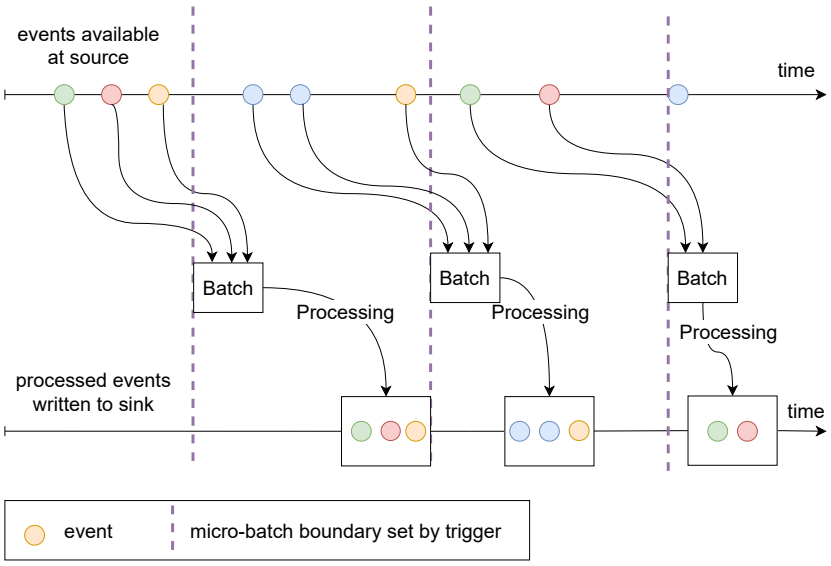


Figure 8. Spark Structured Streaming micro-batch processing with set trigger interval.

Table APIs. The amount of data per batch can be modified when streaming from a file using the *maxfilespertrigger* parameter, for Delta tables in Satabricks both *maxfilespertrigger* and *maxbytespertrigger* parameters can be used. If streaming from Kafka, the *maxoffsetspertrigger* parameter can be used.

Structured streaming does not handle input that is not append-only and when creating a streaming pipeline where one streams' output is used as another streams' input, complete mode can not be used as the first stream as it modifies the underlying Delta table [34].

3.4.2 Output modes

There are three distinct output modes for writing results from a continuously updated *Result table* to external storage in a structured streaming context. These modes are append, complete and update. Append mode is the default mode, where only the rows added to the *Result Table* since the last trigger will be written to external storage. Complete mode writes the entire *Result table* to the external storage and update mode only writes rows, which were updated in the *Result table* since the last trigger to external storage. The different types of output modes have use cases for different types of queries. Append mode requires all queries with aggregation to use a window or group based on a field which is watermarked, so that the aggregation can handle late data. Complete and update mode do not require windows or watermarks, however they require aggregation. Update is not supported in Databricks. Different output modes allow for different results in the

external sinks.

When writing to a sink, the *pyspark.sqlDataFrame.writeStream* method is used to write a streaming DataFrame to external storage systems. The writer has the trigger option, which allows the interval for processing to be set using the *processingTime* option. If processing time is not specified, the micro-batches are generated as soon as the previous micro-batch has completed processing. If the processing time is set, then if the previous micro-batch has completed within the interval, the engine will wait until the next interval to kick off the next micro-batch. If the previous micro-batch takes longer, the next micro-batch will start as soon as the previous one completes.

The *foreacBatch* operation is a feature in Spark Structured Streaming, allowing users to apply arbitrary functions to the output data of every micro-batch in the streaming query, having two primary parameters: the output data as a DataFrame or Dataset and the micro batch's unique ID. This allows each micro-batch to be processed as a static DataFrame, supporting various DataFrame operations, which are generally not feasible with streaming DataFrames. *ForeachBatch* operation inherently provides at-least-once write guarantees, however the output can be deduplicated using the batch ID.

3.4.3 Stateful operations

Stateful operations retain information about the current state of records, which is dependent on previous records, for example a maximum over records with the same id. The retention and constant updating of an alignment is considered stateful. In streaming, unless using complete mode, stateful operations require a time window on a watermarked field. Thus the complete mode can be used to calculate the aggregate over all events and the append mode can calculate the aggregate over a fixed or dynamic period of time, depending on the window operation. The states are stored in the state store - a versioned key-value store, which helps to handle stateful operations across batches. Arbitrary stateful operations can be handled in Spark using Scala or Java, however support for Pyspark has only been added in the past few years as an experimental method. For queries in SQL there is no support for arbitrary stateful operations.

3.4.4 Window operations

It is possible that data is ingested out of order and needs to be streamed taking into account the event time rather than the processing time. In structured streaming, the concept of event time is integrated using watermarks, enabling the system to handle data that may arrive later than expected. Watermarking allows the system to define a threshold for how late the data can arrive. Old aggregates are either maintained or discarded based on the current event time observed in the data stream. This ensures that even late-arriving data can be factored into the calculations without requiring continuous storage of all intermediate states.

In scenarios requiring precise timing and data updates such as window aggregations, the system uses the watermark to decide when to update or discard state information based on the event time of incoming data relative to the established watermark threshold. The formula given for a specific window ending at time T, the engine will maintain state and allow late data to update the state until $(max\ event\ time\ seen\ by\ the\ engine - late\ threshold > T)$.

Spark supports three different types of time windows: tumbling, sliding and session windows, each serving different use cases in data stream management. Tumbling windows are characterised by fixed-size, non-overlapping intervals where each input is mapped to a single window. In contrast, sliding windows, though also fixed in size, allow overlapping intervals if the slide duration is less than the window duration, thereby enabling an input to be associated with multiple windows.

Session windows differ notably from the other types by having a dynamic duration that adjusts based on the timing of incoming data. These windows begin with an input and extend if subsequent inputs are received within a predefined gap duration. The window closes if no further inputs are received within this gap following the most recent input. This behaviour is controlled through the session-window function, which can be configured with static or dynamic expressions to determine the gap duration. Session window requires an additional column in the grouping key aside from the session window, as seen in Figure 9.

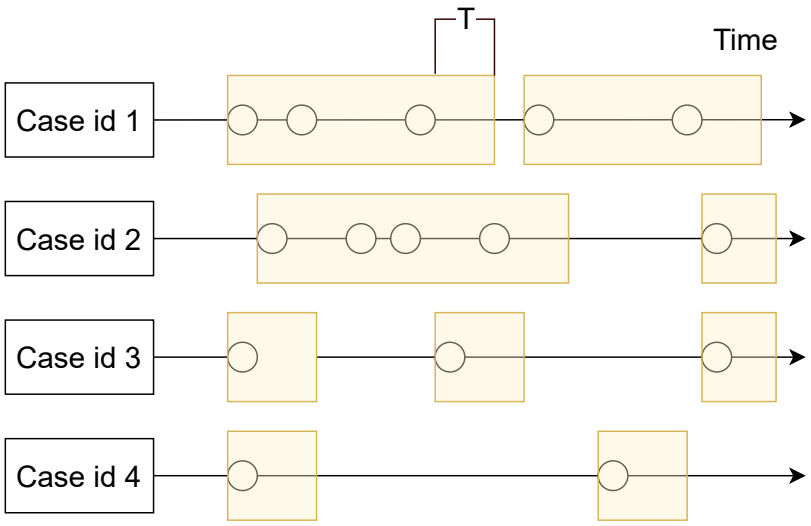


Figure 9. Events grouped by session window on event time and case ID, with fixed gap T.

3.4.5 User defined functions

User defined functions or UDFs in Spark SQL provide a mechanism for users to define their own functions that operate on single rows. These functions are defined using the *UserDefinedFunction* class, which offers several methods to specify the function's properties. UDFs allow creating functions filling specific needs, but come with a cost in execution speed. UDFs can create significant bottlenecks in code execution. Code that executes in the Java Virtual Machine, such as code written in Scala or Java is faster than Python UDFs [35].

3.5 Databricks

Databricks [36] is a cloud-based platform, which provides a managed Spark environment. The platform features an integrated workspace with interactive notebooks, compatible with all Sparks APIs. Spark on Databricks has some limitations, for example not allowing update mode for writing data to sink.

In DataBricks, the default data table format is a Delta table [37]. Delta tables are built on top of the Delta Lake storage layer, which stores data and tables in the Databricks lakehouse. Delta Lake extends Parquet data files with a file-based transaction log providing ACID guarantees. ACID stands for Atomicity, Consistency, Isolation and Durability. Atomicity ensures that all transactions are either fully completed or fail. Consistency means that every transaction leads to a valid state of the database. Isolation ensures that transactions executed concurrently do not affect each other's outcomes. Durability guarantees that the changes made are permanent once a transaction is committed [38].

4 Approach

This section introduces the approach for SQL-based conformance checking in Spark Structured Streaming. The approach used for the continous processing solution is based on the I Will Survive algorithm [4] but has been adapted to accommodate the batch processing method used in Spark. The final implementation presented in this thesis differs from the original algorithm due to these adaptations.

4.1 I Will Survive

I Will Survive (IWS) is an approximate algorithm for conformance checking and finding the optimal alignment. The IWS data model consists of event streams, a state buffer and a trie representing the proxy log. The IWS algorithm expects the events to be processed

one by one in the temporal order. An overview of the IWS approach will be given, as a comparison for the SQL method, as both methods use a trie-based approach.

The algorithm keeps a buffer for events that have yet to be processed, called a state buffer. The state buffer also holds information about events that have been processed. The state buffer consists of the current trie node matching the previous event, current prefix alignment up to the current node, trace suffix containing traces to be processed, cost of the prefix alignment and decay time. The decay time is the duration a state is held in the state buffer and it can be fixed or dynamic. For each new event per case ID, the decay time is decremented.

A look-ahead limit is implemented to handle model moves, as a model move must be deemed to have at least as low of a cost as making a log move. The look-ahead limit is defined as the sum of the size of the trace suffix, the level of the current node plus 1. The algorithm first checks for possible synchronous moves: if none exist, the states are looped to create log and model moves. A model move is only realised if a full substring match to the suffix is obtained in the paths below the state node, such that the first matching node is at most at the level of the look-ahead limit. At first, the children of the current state node are checked; if there are no suffix matches, the look-ahead limit is decreased by one, and the next level of nodes is checked. If there is more than one element in the trace suffix and the look-ahead limit has been exhausted for the current suffix, then the first element of the suffix is removed, and the look-ahead limit is recalculated. In Figure 10 first two synchronous moves are made, however using the look-ahead limit as well as suffix pruning, a model move is made on Q, B is changed to a log move and the suffix $< X, Y, Z, K >$ is matched in the model.

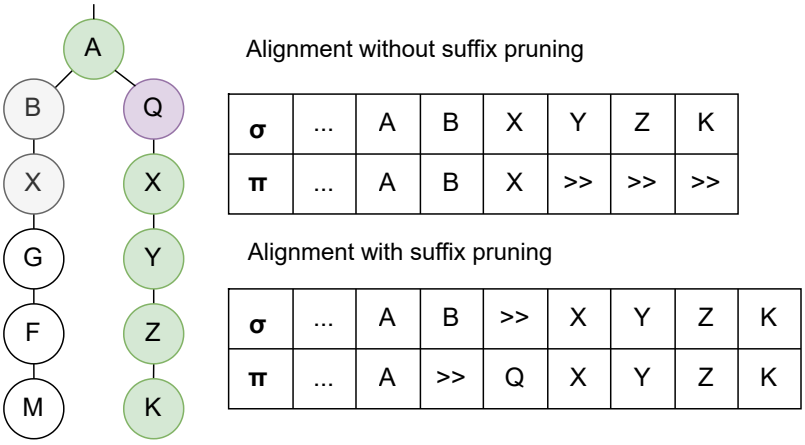


Figure 10. The motivation for suffix pruning.

4.2 Method using Apache Kafka

At first, a solution using Apache Kafka and ksql was tested. A solution using Kafka and Kafka Streams has been proposed [39], but a solution using ksqlDB has not. However, the tables in ksqlDB are designed to hold the latest state by a unique identifier (primary key), and arbitrary state management using only ksql is not possible. Additionally, creating a state table in ksql to be written to and used in the same query is not possible. A possible solution would have been to use a separate relational database as a sink and source for the states, however, as states must be updated before another event can be processed, this was not considered a suitable solution [23].

Another possibility was to create the states as a stream to be inserted into, and create a table from the stream to be joined with the event stream, which would then be inserted back into the state stream. However, this solution was not investigated further as during initial tests of the validity of this solution, the latest state table was not updated before a new event was processed. The structure of this possible solution can be seen in Figure 11. Based on the current running example, if two events a and b from the same case id would be sent in very small intervals, the first event would be processed, then inserted into the state stream and then updating the latest state table, it is possible that event b being processed next is processed before the latest state table can be updated, meaning that both a and b would be processed on an empty state table, without b having up to date information about the state being updated with previous event a.

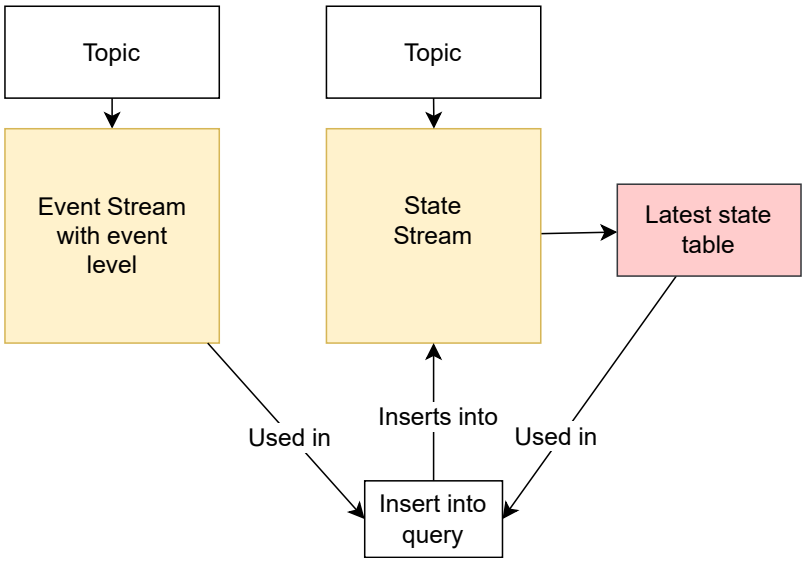


Figure 11. Tested structure using ksql.

4.3 Method using Spark

This section describes the solutions and limitations of using Spark. Spark was chosen as the second method to experiment with, as structured streaming is built upon the SQL API. The experiments with Kafka and ksql solidified two essential problems: firstly, referencing the states table in the same streaming query that updates the states table must be possible. Secondly, the state table must be updated before processing a new event. An overview of the proposed structure of the stream conformance checking is outlined in Figure 12.

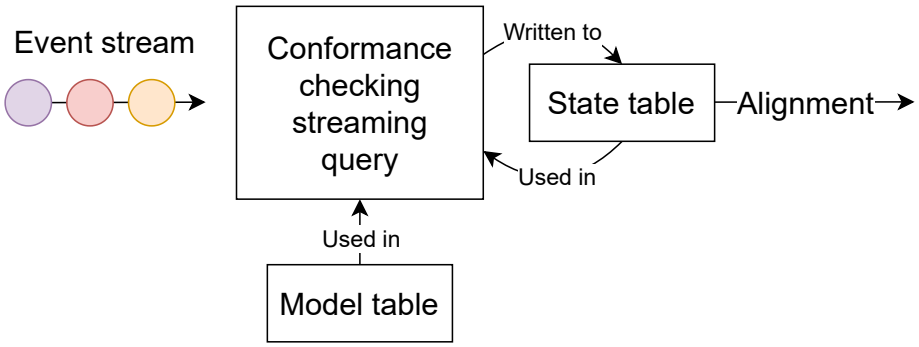


Figure 12. The general setup of the solution.

4.3.1 Pre-processing

The model and label tables were constructed off-line before running the conformance checking stream using the PM4PY library.

The same input datasets were used as in the IWS solution, with 2000 generated traces and the maximum number of loops set to 3. The trie and the label lookup tables were stored in separate tables, as described in table 4 and table 5.

Table 4. Model table

node_id	String	
label	String	
level	Integer	
n_depth_children	Array of Structured fields:	
	node_id	String
	label	String
	level	Integer
	events_between	Array of Strings

Table 5. Label table

event	String
label	String

The model table consists of four columns. The node-id, a string field represents the prefix of an activity in the model. The model table label field references a unique label per activity, where labels in alphabetical order are generated for each activity, in the order they first appear in the proxy log. The level field shows the level of the node and the n-depth-children holds all children of the nodes with levels below the fixed limit to look ahead in the model. The n_depth_children field is calculated during the model creation process and can be used in the solution, instead of joining with the model table using a pattern matching join as these joins are computationally expensive in Spark. The node ID is kept as a string value and is used in the queries to perform substring matches and on joins.

The label table is used as a look-up for the incoming events, to find the labels associated with each event, which match the labels in the model table

4.3.2 Continuous processing

Initially, using continuous processing was considered. A new solution had to be created, as the IWS algorithm, which uses a while loop to calculate model moves, would not be feasible to implement in Spark SQL. This new solution is explained in section 5 of this thesis. However, using continuous processing as a solution in this thesis is currently not viable in Spark Structured Streaming due to limited query sources and query sinks. Using the continuous processing solution, the only approved source or sink would be Kafka, but it is not guaranteed that the state table would be updated from the sink topic before a new record is processed. This solution was included in this thesis as its logic could potentially be implemented in other SQL-based streaming frameworks or integrating it into the micro-batch based approach, where the batch has only one event per unique case ID.

4.3.3 Micro-batch processing

Conformance checking using micro-batches is a window-based model approach. Only the most recent events bound to a window are stored and processed [2, 16]. However in the approach of this thesis, it is important that the windows do not overlap and that the state data is updated before a new window can start processing. It is possible to reference state data in Spark architecture by writing the streaming DataFrame outputs to a table, from which a temporary view containing the best current alignments can be created. This

view can be accessed in subsequent batches, with the advantage that updates to the view are instantaneously available for querying in the next batch cycle as seen in Figure 13.

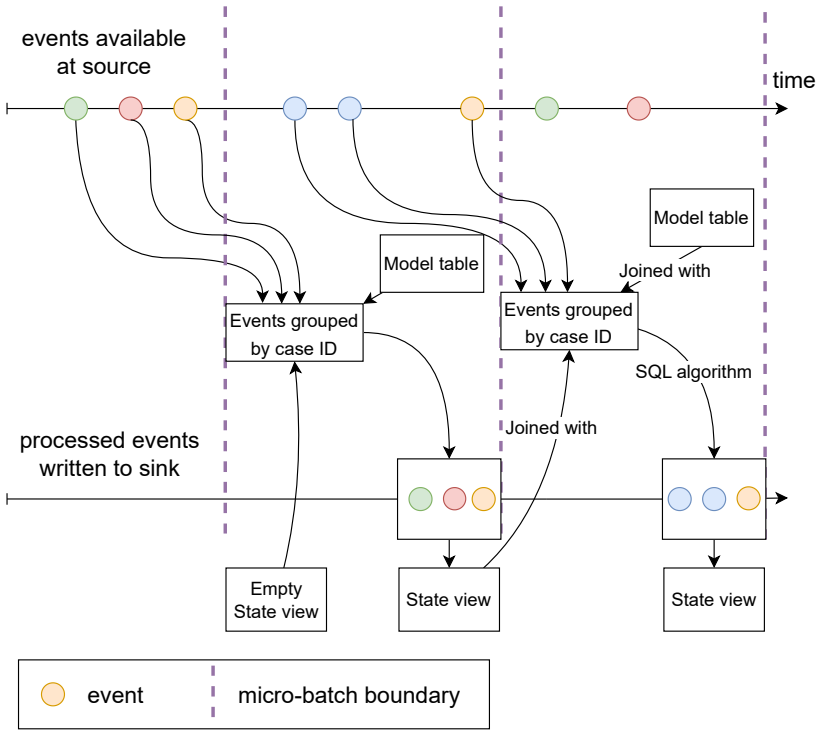


Figure 13. In depth illustration of batch-streaming solution.

However, difficulties arise due to the occurrence of multiple events sharing the same case ID within a single batch coupled with the lack of support for SQL procedures or functions in Spark SQL, complicating sequential processing. A new solution was created to complete model moves and process all events with the same case ID in the batch.

4.3.4 Sequence alignment

As the batch processing solution requires processing a sequence of traces against possible trie branches, the problem recedes to a sequence alignment problem, which is widely researched in bioinformatics. However, most of these algorithms are based on dynamic programming and must be implemented using an UDF.

An UDF was implemented using the Needleman-Wunsch [40] algorithm, proposed by Needleman and Wunsch in the 1970s. It is an algorithm using the dynamic programming technique used freuently in bioinformatics to align protein or nucleotide sequences globally. In the algorithm, an alignment matrix is constructed, which evaluates matches,

mismatches and gaps between sequences, in our case the node-id-s and traces currently being processed. The matches, mismatches and gaps can be thought of as sync, log and model moves. A solution using this algorithm has been proposed here [41].

First the string values are aligned with the model values on the Y-axis and trace activities on the X-axis. The matrix values are calculated for each cell, based on three possibilities - diagonal being match or mismatch, a vertical move representing a skip in the model and horizontal being a skip in the trace. Afterwards the matrix is traversed backwards, moving diagonally for a match and vertically or horizontally to create log and model moves.

Table 6. Needleman-Wunsch algorithm matrix

		A	C	E	D
	0	1	2	3	4
A	1	0	1	2	3
B	2	1	2	3	4
D	3	2	3	4	3
E	4	3	4	3	4

For example, in table 6 the subsequence of the node id to be matched is $A-B-D-E$ and the incoming traces are A, C, E, D. Mismatches are not possible and gap moves cost 1, otherwise the cost is 0. In table 7 the complete alignment based on the matrix has been created. The total number of gap moves on the matrix is 4, the same as the cost of the model if log and model moves both cost 1. This cost is also reflected in the bottom right corner of the matrix.

Table 7. Needleman-Wunsch example alignment

σ	A	C	$\gg$	$\gg$	E	D
π	A	$\gg$	B	D	E	$\gg$

This solution is suboptimal, as UDFs are not optimised in Spark and the Needleman-Wunsch algorithm has the space complexity of $O(mn)$, where m and n represent the number of columns and rows in a matrix, respectively.

For all the implemented solutions, the cost for log and model moves can be set separately.

This thesis explores a two continous processing based solutios in Kafka, ksql and Spark Structured Streaming, and three possible window based solutions using micro-batch processing.

5 Implementation

Five [42] possible solutions were created, one based on a continuous processing model and the other on the batch processing model. For the batch processing model three methods of collecting the activities into one group per case ID per batch were considered, from which the final solution was chosen.The main idea derived from the IWS model was the state buffer. The state buffer is implemented as a state table with the following structure:

Table 8. State table

case_id	String
time_stamp	Timestamp
current_node	String
current_id	String
cost_of_alignment	Integer
previous_events	String
event_level	String
current_node_level	String
decay_time	String

Table 9. Continuous processing mode

trace	String
execution_sequence	String

Table 10. Batch processing mode

Alignment	Array of structs	
	event	String
	move_type	String

The state view can be compared to the housekeeping part of the IWS algorithm, as it filters all states in the state table to save the rows which have the least cost for the current-id and where the decay time is not 0. The solution supports all event inputs, as long as the input data is in the correct correct format, consisting of the fields: "case_id", "event name" and "timestamp". From the input streaming DataFrame, a temporary view called events is created, which is used as the streaming source for the streaming queries.

5.1 Continuous Processing solution

Although a solution has been developed, it has yet to be tested due to its lack of application in Spark Structured Streaming. Nevertheless, it can still be utilized by creating a buffer to process one case ID at a time. This option was included as a potential solution due to the evolving nature of the Spark Structured Streaming API. The solution consists of two steps: first, joining previous state values with new events, and second, processing the possible moves in the alignment.

To begin, the Stream query is initiated, and an event is deemed ready for processing. The labels view is joined to obtain the event label, and the latest-state view is joined on the case ID to obtain the latest states. If the event is the first event for the particular case ID, default initial values are used in place of the actual values, as the values from the state table come later and are concatenated with the current event values. The model table is joined on the current node ID, and if the event is the first event for the case ID, the join is made on the root node.

Next, the moves are processed. The logic for model and sync moves are similar, as all values in the n-depth-children array that have a matching label are selected. This is done to imitate the trace suffix looping in the IWS algorithm, where a model move might be the most optimal move. However, implementing the logic of selecting only model moves which are at least as optimal as log moves was proven to be difficult, as it is not possible to loop in Spark SQL. So all of the possible sync, model, and log moves for an event are selected when the event is processed. Additionally, the look-ahead limit is fixed to 2 in this example for computational considerations.

This solution does not depend on calculations to get an alignment, but rather appends values to the trace and execution sequence string for every incoming event. The view cleans up the states and removes any possible duplicate node IDs, leaving only the alignment with the lowest cost. While this solution is more exhaustive than the IWS algorithm, keeping more states in the state view, the states saved in the view can be limited using either their node level, a maximum node level based on the alignment with the lowest cost, or restructuring the query not to update states with every move. Furthermore, synchronous and asynchronous moves can be split so that asynchronous moves are only performed if there are no synchronous moves. The previous states would remain in the state view, so any incoming event would still calculate model moves based on previous states. Although further optimizations were not discussed, they could be implemented in other SQL-based streaming frameworks, as this solution is not currently possible using Spark Structured streaming.

5.2 Micro-batch processing solution

The initiation process for micro-batch processing is similar to the continuous processing solution. However, the calculation process for the alignment is different, as a sequence

of events must be processed at once, and the previous values of the trace can not be taken into account.

5.2.1 Different methods of grouping data

When processing the event data in a batch query sequentially, the data must be ordered by timestamp and grouped by case ID. To solve this, the functions *array_sort* and collect-list of *Struct(time-stamp, label)* were used to create a sorted array of structs. The final selected solution uses the *forEachBatch* sink. At first, the *forEachBatch* sink was not preferred, as it is a Spark-specific function and can not be implemented like window-based functions in other SQL-based streaming platforms. Additional logic must be implemented to deduplicate data and create a solution with at-least-once write guarantees.

A solution using windows was first experimented with, as using windows allows accounting for late data. However, using a window-based approach with timestamps was not deemed suitable, as Spark outputs the values to the sink based on the watermarked field. So, even when using a session window, it would be possible that the state table would have two window groupings appended to the output sink at the same time, as illustrated in Figure 14.

Furthermore if multiple windows per the same case ID are processed at the same time, then they all use the same version of the state view not yet updated by the newest window. So, a solution based on a session window grouping would only be possible if the delay threshold was 0 seconds, as any incoming event not within the session window would trigger a write to the sink.

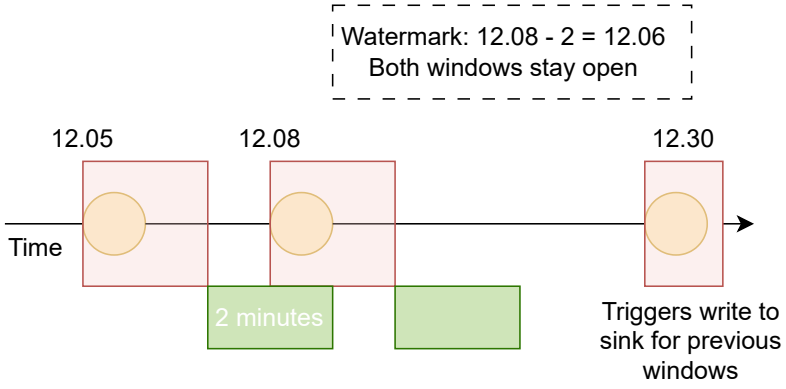


Figure 14. Why watermarking creates issues with sequential processing.

Another possible solution was using the complete output mode, constantly updating the grouped values of the same case ID and getting the unprocessed values from slicing

the array based on the event level. However, not only is the grouped array of labels constantly reordered, but any late data would change the order of the array indexing based on which the current events are chosen.

The final solution chosen was to use the *forEachBatch* sink and create a temporary view of the batch data inside the function to replace the streaming events view. This changes the solution from being a streaming solution to a batch solution. Still, the batch solution can be considered stream processing since it operates within a streaming context.

5.2.2 Solution

The function used in the *forEachBatch* sink, is initiated with an incoming batch. Inside the function, the batch data event view is created. Then a SQL query is initiated, creating a DataFrame, later written to a Delta table. Inside of the query, the data in the batch event view is grouped by case ID and ordered by timestamp into arrays with the label and timestamp. Then the data is joined with the state view on case ID and expanded, so that every array event is separated to its own row, with its index in the trace additionally. Then the values are joined with the model table on label and depth based on each event's index in the trace where model node level is less than $current\ node\ level + index\ in\ current\ array + N + 2$. Thus, for every event N number of nodes from it's starting point are assessed. This is not optimal, as it assumes that the previous values will be sync moves, however the search must be limited for better performance. After all possible nodes are selected, the results are filtered to only get the nodes for the last M events in the array, where M is the decay time, as if the moves were sync moves they would be reflected in the node ID, if not, they would be removed based on the decay time.

Additional transformations were tested, to limit the amount of times the UDF would be called, such as filtering the different prefixes and only calculating one alignment per unique suffix, however cross-joins create a overhead and might not be optimal.

To keep track of the cost of the alignment, the UDF outputs every move with its cost. When the previous node ID-s and the new node ID-s are filtered, then the UDF is applied. The UDF algorithm is implemented in Scala [42] based on an existing implementation in Java [43].

6 Assessment of the micro-batch solution

The performance of the selected method was not thoroughly assessed, due to time constraints. However, the resulting alignments of the model were validated during the creation of the solutions using the same input data on the IWS algorithm [44].

The performance was assessed on a trie with the maximum depth of 344 and node count of 9799, from the BPI 2012 1k sample simulation file. The used log files for testing

was the BPI 2012 2k 0.95 noise sample simulation the log file. The log file contains only 8026 events, and was calculated in one batch of 2000 traces, in 39 seconds. The sum of the cost of all optimal alignments was 753. However, if data was split into smaller parts, and limiting the files per trigger the overhead of reading from small files was too large for the stream to start processing. When inserting data to a static table, from which a streaming DataFrame was created, the stream was able to process the data in small batches.

For this thesis the community edition of Databricks was used, which is hosted on Amazon Web Services and offers 15 GB clusters, a cluster manager and the notebook environment.

7 Future work

In this thesis, various methods of conformance checking using SQL streaming solutions were implemented. Moving forward, a detailed comparative analysis of these methods would be beneficial, as well as performance analysis of the existing method. This analysis would shed light on the performance differences among the streaming solutions.

The chosen *forEachBatch* solution could also be improved by adding additional logic to minimise the amount of times the alignment must be calculated using the UDF. Additionally, an in-depth analysis of how to implement logic similar to IWS, so that that model moves are only conducted if they cost the same amount or less as making log moves. As the initial solution was created with the window function inside a streaming query, it was not optimized to utilise transformations that are not possible within a streaming query but are feasible within a batch DataFrame when using *forEachBatch*. For example investigating whether a fully SQL-based solution would be possible to implement using cross-joins to create all possible values of a trace suffix. However, research on in what possible situations cross-joins would be more effective and less expensive than using the UDF should be conducted.

One key area that needs enhancement in the solutions implemented, is their ability to manage late-arriving or out-of-sequence data. An additional grouping query based on time windows at an earlier stage in the data processing pipeline could be introduced to improve this. These time windows could then be aggregated in the downstream state calculation query to enhance data consistency and accuracy.

Furthermore, it would be worthwhile to investigate the potential of implementing a continuous processing query using another platform, which supports SQL, but also sequential processing of events. Another direction could be to develop a solution in Spark Streaming, utilising either built-in Scala or Python methods for state management and not focus on creating a solution using SQL. With the PySpark API continually evolving, this approach might unlock new functionalities that could be useful in creating a solution.

8 Conclusion

The objective of this thesis was to experiment creating a conformance checking solution using a trie structure with different streaming platforms that support SQL. The two platforms researched were Apache Kafka with ksqlDB and Apache Spark, using Spark's Structured Streaming API.

An overview of the theoretical background of process mining, conformance checking, Spark Streaming and Kafka and ksql were presented, as well as describing implemented solutions.

At first a solution on Apache Kafka was tested, however, it was deemed not feasible to store already calculated alignments in a way that incoming events would be updated on the latest data. From testing a solution using Apache Kafka, two critical nuances were pinpointed: the alignment calculated on previous data must be updated before the arrival of new events and events must be processed either in order or all in the same window, as multiple concurrent updates of alignments with the same case ID will cause the alignments to be calculated without correct state reflections. Furthermore, processing data in a streaming context limits the possibilities of transformations on the data. As these technical problems were researched thoroughly, the final implementation of the conformance checker was not well-optimised and a performance comparison between the results was not completed.

The final solution proposed was based on a micro-batch solution, where data was grouped by case ID and a sequence alignment algorithm was implemented in Scala to calculate the most optimal alignment. The following methods of improvement were discussed: testing the continuous processing model on other streaming platforms, updating the current proposed solution and handling late arriving data.

References

[1] A. Burattin, "Streaming process discovery and conformance checking," in *Encyclopedia of Big Data Technologies*, S. Sakr and A. Zomaya, Eds. Springer International Publishing, 2018, pp. 1–8. [Online]. Available: http://link.springer.com/10.1007/978-3-319-63962-8_103-1

[2] ——, "Streaming process mining," in *Process Mining Handbook*. Springer International Publishing, 2022, vol. 448, pp. 349–372, series Title: Lecture Notes in Business Information Processing. [Online]. Available: https://link.springer.com/10.1007/978-3-031-08848-3_11

[3] A. A. Adriansyah, "Aligning observed and modeled behavior," 2014, publisher: [object Object]. [Online]. Available: https://research.tue.nl/en/publications/aligning-observed-and-modeled-behavior(6eac4d54-1d66-4161-80c3-6e95696ea87f).html

[4] K. Raun, R. Tommasini, and A. Awad, "I will survive: An event-driven conformance checking approach over process streams," in *Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems*. ACM, 2023, pp. 49–60. [Online]. Available: https://dl.acm.org/doi/10.1145/3583678.3596887

[5] S. J. Van Zelst, A. Bolt, M. Hassani, B. F. Van Dongen, and W. M. P. Van Der Aalst, "Online conformance checking: relating event streams to process models using prefix-alignments," vol. 8, no. 3, pp. 269–284, 2019. [Online]. Available: http://link.springer.com/10.1007/s41060-017-0078-6

[6] D. Schuster and S. J. Van Zelst, "Online process monitoring using incremental state-space expansion: An exact algorithm," in *Business Process Management*, D. Fahland, C. Ghidini, J. Becker, and M. Dumas, Eds. Springer International Publishing, 2020, vol. 12168, pp. 147–164, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-030-58666-9_9

[7] Grammarly, accessed (15.05.2024). [Online]. Available: https://www.grammarly.com/

[8] "Cambridge dictionary." [Online]. Available: https://dictionary.cambridge.org/dictionary/english/process,

[9] N. Graves, I. Koren, and W. M. Van Der Aalst, "ReThink your processes! a review of process mining for sustainability," in *2023 International Conference on ICT for Sustainability (ICT4S)*. IEEE, 2023, pp. 164–175. [Online]. Available: https://ieeexplore.ieee.org/document/10292162/

[10] "Process mining in action: Principles, use cases and outlook," 2020. [Online]. Available: http://link.springer.com/10.1007/978-3-030-40172-6

[11] W. M. P. Van Der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Berlin Heidelberg, 2011. [Online]. Available: https://link.springer.com/10.1007/978-3-642-19345-3

[12] A. Vaisman, "An introduction to business process modeling," in *Business Intelligence*, M.-A. Aufaure and E. Zimányi, Eds. Springer Berlin Heidelberg, 2013, vol. 138, pp. 29–61, series Title: Lecture Notes in Business Information Processing. [Online]. Available: https://link.springer.com/10.1007/978-3-642-36318-4_2

[13] H. M. Marin-Castro and E. Tello-Leal, "Event log preprocessing for process mining: A review," vol. 11, no. 22, p. 10556, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/22/10556

[14] W. Van Der Aalst, A. Adriansyah, and B. Van Dongen, "Replaying history on process models for conformance checking and performance analysis," vol. 2, no. 2, pp. 182–192, 2012. [Online]. Available: https://wires.onlinelibrary.wiley.com/doi/10.1002/widm.1045

[15] O. Etzion, "Event stream," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer US, 2009, pp. 1063–1063. [Online]. Available: http://link.springer.com/10.1007/978-0-387-39940-9_590

[16] "Process mining handbook," 2022. [Online]. Available: https://link.springer.com/10.1007/978-3-031-08848-3

[17] J. Carmona, B. Van Dongen, and M. Weidlich, "Conformance checking: Foundations, milestones and challenges," in *Process Mining Handbook*, W. M. P. Van Der Aalst and J. Carmona, Eds. Springer International Publishing, 2022, vol. 448, pp. 155–190, series Title: Lecture Notes in Business Information Processing. [Online]. Available: https://link.springer.com/10.1007/978-3-031-08848-3_5

[18] W. L. J. Lee, H. Verbeek, J. Munoz-Gama, W. M. Van Der Aalst, and M. Sepúlveda, "Recomposing conformance: Closing the circle on decomposed alignment-based conformance checking in process mining," vol. 466, pp. 55–91, 2018-10. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0020025518305413

[19] M. Crochemore and T. Lecroq, "Trie," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. Springer US, 2009, pp. 3179–3182. [Online]. Available: http://link.springer.com/10.1007/978-0-387-39940-9_1143

[20] xes-standard website, accessed (15.05.2024). [Online]. Available: http://www.xes-standard.org./

[21] History of xes, accessed (15.05.2024). [Online]. Available: https://www.tf-pm.org/resources/xes-standard/about-xes/role-and-history

[22] PM4py, accessed (15.05.2024). [Online]. Available: https://pm4py.fit.fraunhofer.de/documentation

[23] ksqldb documentation, accessed (15.05.2024). [Online]. Available: https://docs.confluent.io/platform/current/ksqldb/overview.html

[24] Apache spark website, accessed (15.05.2024). [Online]. Available: https://spark.apache.org/

[25] E. Shaikh, I. Mohiuddin, Y. Alufaisan, and I. Nahvi, "Apache spark: A big data processing engine," in *2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*. IEEE, 2019, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/8988541/

[26] Spark documentation: Cluster mode overview, accessed (15.05.2024). [Online]. Available: https://spark.apache.org/docs/latest/cluster-overview.html

[27] N. Ahmed, A. L. C. Barczak, M. A. Rashid, and T. Susnjak, "An enhanced parallelisation model for performance prediction of apache spark on a multinode hadoop cluster," vol. 5, no. 4, p. 65, 2021. [Online]. Available: https://www.mdpi.com/2504-2289/5/4/65

[28] "Databricks presentation: Building a modern application w/ DataFrames, , accessed (15.05.2024)." [Online]. Available: https://www.slideshare.net/databricks/building-a-modern-application-with-dataframes-52776940

[29] Spark SQL, DataFrames and datasets guide, accessed (15.05.2024). [Online]. Available: https://spark.apache.org/docs/latest/sql-programming-guide.html

[30] Lydia Parziale, Joe Bostian, Ravi Kumar, Ulrich Seelbach, and Zhong Yu Ye, *Apache Spark Implementation on IBM z/OS*, 2016. [Online]. Available: https://www.redbooks.ibm.com/redbooks/pdfs/sg248325.pdf

[31] Structured streaming programming guide, accessed (15.05.2024). [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[32] J. Torres, M. Armbrust, T. Das, and S. Zhu. (2018) Introducing low-latency continuous processing mode in structured streaming in apache spark 2.3. [Online]. Available: https://www.databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html

[33] Spark streamexecution class GitHub source, accessed (15.05.2024). [Online]. Available: https://github.com/apache/spark/blob/master/sql/core/src/main/scala/org/apache/spark/sql/execution/streaming/StreamExecution.scala

[34] Databricks documentation on delta lakes, accessed (15.05.2024). [Online]. Available: https://docs.databricks.com/en/structured-streaming/delta-lake.html

[35] (2024) Databricks doumentation: What are user-defined functions (UDFs)?, accessed (15.05.2024). [Online]. Available: https://docs.databricks.com/en/udf/index.html

[36] DataBricks website, accessed (15.05.2024). [Online]. Available: https://www.databricks.com/

[37] Databricks community edition FAQ, accessed (15.05.2024). [Online]. Available: https://www.databricks.com/product/faq/community-edition

[38] What are ACID guarantees on azure databricks, accessed (15.05.2024). [Online]. Available: https://learn.microsoft.com/en-us/azure/databricks/lakehouse/acid

[39] D. Schuster and G. J. Kolhof, "Scalable online conformance checking using incremental prefix-alignment computation," in *Service-Oriented Computing – ICSOC 2020 Workshops*, H. Hacid, F. Outay, H.-y. Paik, A. Alloum, M. Petrocchi, M. R. Bouadjenek, A. Beheshti, X. Liu, and A. Maaradji, Eds. Springer International Publishing, 2021, vol. 12632, pp. 379–394, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-030-76352-7_36

[40] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," vol. 48, no. 3, pp. 443–453, 1970. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0022283670900574

[41] N. An, Z. Wenyu, S. Leo, and E. Bjoern, *Conformance Checking for a Medical Training Process Using Petri net Simulation and Sequence Alignment*, 2020. [Online]. Available: https://www.researchgate.net/publication/344827979_Conformance_Checking_for_a_Medical_Training_Process_Using_Petri_net_Simulation_and_Sequence_Alignment

[42] Repository of code for the experiments, accessed (15.05.2024). [Online]. Available: https://github.com/Annilo/thesis

[43] NeedlemanWunsch in java, accessed (15.05.2024). [Online]. Available: https://github.com/FrancisLawlor/NeedlemanWunsch/tree/master

[44] Iws streaming github, accessed (15.05.2024). [Online]. Available: https://github.com/DataSystemsGroupUT/ConformanceCheckingUsingTries/tree/streaming

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Agnes Annilo**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Exploring SQL-based near real-time conformance checking using Stream Processing Engines,

   supervised by Kristo Raun.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Agnes Annilo
15/05/2024