University of Zurich[UZH]

MSc Thesis 29.03.2018

Exploring Context-aware Stream Processing

Benedikt Bleyer
of Waldkirch BW, Germany

Student-ID: 14-706-162
benedikt.bleyer@uzh.ch

Advisor:
Daniele Dell'Aglio, PhD

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I want to express my sincerest gratitude to my supervisor Daniele Dell'Aglio, PhD for his competent and enthusiastic support during this thesis. With his immense knowledge and passion in the area of stream processing and stream reasoning, he was able to help me during all times of research, implementing and writing this thesis. In addition, I would like to thank Assistant Professor Alessandro Margara, PhD for his active support, domain expertise and programming advice.

I would also like to thank Professor Abraham Bernstein, PhD and the members of the Dynamic and Distributed Information Systems Group for giving me the opportunity to work on such an interesting topic.

Further, I would also like to thank Jonas Traub for his advice regarding Apache Flink and the soccer monitoring case study as well as Andreas Gruhler and Christian Thol for proofreading my thesis.

Last but not least I would like to thank my friends and family for their constant support and encouragement.

Zusammenfassung

Unternehmen, Privatpersonen und Sensoren produzieren heutzutage kontinuierlich Daten, daher sollte auch die Verarbeitung dieser Daten in kontinuierlicher Art und Weise erfolgen. Eine steigende Anzahl von Anwendungsfällen für Data Streaming benötigt Modelle und Systeme, welche sich dynamisch an Umweltzustände anpassen können und fähig sind verschiedene Informationstypen wie Kontext, Fakten und Hintergrundinformationen zu integrieren um gewinnbringende Erkenntnisse in Echtzeit zu liefern. Diese Thesis präsentiert mit dem "Context-aware, Facts and Background integrated dynamic Stream Processing" (CoFaBidSP) ein entsprechendes konzeptionelles Modell. Die Ergebnisse der Evaluation des implementierten Prototyps zeigen, dass der vergrösserte Funktionsumfang zu nahezu keinen Performance Einbussen bei Laufzeit und Verarbeitungskapazität führt.

Abstract

Today's data is continuously produced by companies, private people and sensors, therefore processing the data should also be in a continuous way. An increasing number of use cases for streaming data require models and systems which can adapt their processing based on changes in the application context and need to be able to integrate various information types such as context, facts and background information to deliver valuable near real time insights. This thesis proposes a model for Context-aware, Facts and Background integrated dynamic Stream Processing (CoFaBidSP). The evaluation results for the implemented prototype show that the metrics run time and events per seconds remain nearly constant, even by including more functionality such as the integration of various information types in dynamic stream processing.

Table of Contents

1

Introduction

Today's data is continuously generated and processed by companies, private people and machines (including e.g. sensors or computers). Such entities are very different, and the data they produce varies in structure, volume, variety, velocity and veracity (IBM, 2013). Traditionally, data is processed and stored in a Database Management system (DBMS) and then prepared for data analysis. This procedure is well-researched and widely applied in practice. The amount of data is known ex ante and new data is processed periodically in batches, e.g. in data warehouses.

Even in the era of big data the processing of the data is done periodically, but in a distributed fashion, e.g. by using the MapReduce model (Dean and Ghemawat, 2008). A general characterization of a distributed system is given by Tanenbaum and Van Steen (2007, p. 2): "A distributed system is a collection of independent computers that appears to its users as single coherent system." It is important that distributed systems usually use different types of transparency such as access, location, migration, relocation, replication, concurrency and failure transparency. Those transparency types are also used to achieve high scalability by distributing data and its processing to independent computers. (Tanenbaum and Van Steen, 2007, p. 5ff.)

Also the Web is a distributed system. It is heavily used by companies and private people to produce, sell and buy products as well as building (social) networks e.g. to communicate and collaborate. The corresponding data is produced continuously and also should be processed in that way. Such data, which we refer to as *data stream* has the following main characteristics: (i) The data elements are produced continuously, (ii) the arrival order is not known, (iii) the size is potentially unbounded and (iv) its processing of the data elements should be near real-time (Babcock, Babu, Datar, Motwani, and Widom, 2002, p. 2; Cugola and Margara, 2012, p. 6).

In the field of data streams, there are two main system paradigms: Data Stream Management Systems (DSMS) and Complex Event Processing systems (CEP) (Cugola and Margara, 2012, p. 2ff.). DSMSs focuses on processing data streams from various sources and emitting output streams for further analysis. CEP systems on the other hand handle the elements of a data stream as events. They analyze the incoming events to detect patterns and take actions on a higher-level of abstraction, e.g. smoke detector. For

both paradigms there exist centralized as well as distributed systems implementations. (Cherniack et al., 2003; Helmer, Poulovassilis, and Xhafa, 2011)

Due to the unbounded nature of data streams, it is impossible to give a final answer for a specific query. Therefore, windows are usually used to partition the incoming data stream by time or element frequency and to optimize the necessary processing resources. A fixed window would include all the data elements of the last five minutes or the last ten data elements. Sliding windows use a similar approach, but those windows are overlapping, e.g. a window is generated for the last hour every 30 minutes. (Akidau et al., 2015; Cugola and Margara, 2012, p. 21) For capturing rapid changes in the values of events or aggregating events of user sessions those windows can not be used.

Further approaches for constructing windows based on the content of the data streams are proposed by Grossniklaus, Maier, Miller, Moorthy, and Tufte (2016) and Akidau et al. (2015). Frames (Grossniklaus et al., 2016) are based on the value of data elements, e.g. a frame is created if a value of an attribute of the data stream reached a specific threshold. The Data Flow Model (Akidau et al., 2015) introduces session windows to examine the click stream of customer on a shopping website. The size of such a key-based data stream is bounded, but it differs from customer to customer and is not know ex ante.

An addition to these approaches can be done by including the application context (related terms are state and situation) of the real world in the system. Application contexts are defined as "real-world higher-order situations the duration of which is not known at their detection time and potentially unbounded" (Poppe, Lei, Rundensteiner, and Dougherty, 2016, p. 415). It can be used as an (resource) optimization technique and to extend the possibilities for analytics. Examples for contexts in the area of network monitoring could be "underloaded", "overloaded" or "crashed" (Poppe et al., 2016). Possible context types for an soccer game could be "1st half","2nd half", "Break". In order to adjust advertisement, recommendations or processes during the soccer game, the visitor's overall satisfaction as well as different statistics about ball possession, shots on goal, player performance are relevant. In addition to the context, stable background knowledge as well as facts with specific validation periods should be combined with the incoming data elements of the stream. Kietz, Scharrenbach, Fischer, Bernstein, and Nguyen (2013) proposed TEF-SPARQL, a query language for time annotated event and fact Triple-Streams. A conceptual model for separating the state (context) from stream processing components is provided by Margara, Dell'Aglio, and Bernstein (2017).

The research questions that I investigate in this Master thesis are the following:

- How could a conceptual model for stream processing look like, which allows the combination of context-awareness, background knowledge and time-restricted facts?
- What would be the architecture of an engine implementing such a model?

The rest of the MSc thesis is organized as follows: Chapter 2 gives a summary of the related work and compares the existing methods, approaches and models. After in-

troducing different motivating case studies, Chapter 3 provides details of a conceptual model to support those case studies. Chapter 4 focuses on the technical implementation of the conceptual model with the technical components and their interaction. Evaluation results are provided in Chapter 5. After discussing limitations and future work in Chapter 6, I will conclude with Chapter 7.

2

Related Work and Background

This chapter summarizes relevant studies in the context of stream processing, followed by a state of the art analysis of existing models, approaches and systems related to stream processing including context-awareness, background knowledge and facts (Section 2.2).

The following section will introduce different styles and/or requirements for queries as well as two notions of time relevant for stream processing. Based on that the concept of windows is described.

2.1 Query Types and Windowing in data stream processing

Two distinction of queries can be done, the first one is between one-time queries and continuous queries (Babcock et al., 2002, p. 2). *One-time queries* are created and evaluated exactly once over the available data at this point in time. *Continuous queries* are evaluated multiple times. Each result will reflect the data elements received so far in the data stream.

Predefined queries and Ad-hoc queries are the second distinction (Babcock et al., 2002, p. 2). *Pre-defined queries* are created before the data stream is received and are usually continuous queries. *Ad-hoc queries*, on the other hand, are created after the data stream is already emits events, here it is then important to consider carefully what range of data element is important for evaluating the query. An ad-hoc query could be either one-time or continuous.

Furthermore, it is important on what underlying data structure the query is evaluated. The data elements of the data streams can be structured in a key-value format, Resource Description Framework (RDF) graphs or relational tables. RDF is "a data model in which the basic unit of information is known as a triple. A triple consists of a subject, a predicate and object." (DuCharme, 2013, p. 24) The data of the triples can be represented in a directed graph. In this graph "each subject or object value is a labeled node [...] and the predicates are the labeled arcs connecting the nodes." (DuCharme, 2013, p. 33) Another aspect is the provided operators of the query language used to create the queries, like projection, filter, aggregations. Based on relational algebra different languages are available at the moment: SPARQL used for queries on the semantic web,

CQL (Continuous Query Languages) or also TEF-SPARQL (which can be used to query fact time annotated triples).

Due to the unbounded nature of streaming data, query languages for DSMS introduced a new window operator to deliver approximate query answers (as described in Babcock et al., 2002, p. 6f.). Currently three types of windows can be distinguished: First, Windows based on external, fixed criteria like count or time (Babcock et al., 2002, p. 7f.). For example, a count-based fixed or tumbling window with size 50 contains always 50 events. A fixed window with size of 10 minutes consists of the events of the last 10 minutes, see Figure 2.1. A sliding window is "defined by a fixed length and a fixed period."(Akidau, 2015). With a time period of 5 minutes and a fixed length of 10 minutes, every five minutes a window with a fixed length of 10 minutes is created, which means that the windows are overlapping by five minutes. Second, windows based on the data content like frames (Grossniklaus et al., 2016) (see Section 2.2) or session windows (Akidau et al., 2015, p. 1794), see Section 2.2. Third, windows which are aware of higher-level contexts, see Section 2.3. (Poppe et al., 2016).

Another important notion in DSMS is time: At what time the event is generated

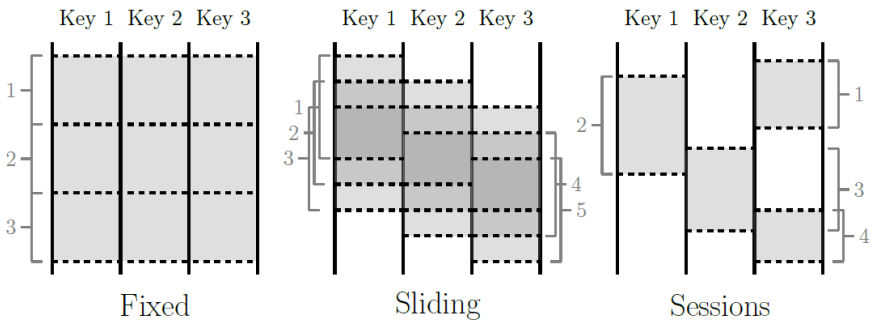


Figure 2.1: Window patterns (Source: Akidau et al., 2015, p. 1794)

(event time) by the sensor or when the event is processed by the stream processing system (processing time). While the event time can never change, the processing time can evolve depending on the processing flow in the stream processing engine. Due to various reasons the processing time and event time differ, which means that events are placed in different windows depending on what timestamps are used, event time or processing time. Figure 2.2 shows the time domain skew. If the event time and processing time would be the same, the straight gray dotted line would be true. But often the processing is delayed. The black arrow shows the difference the the processing and event time.

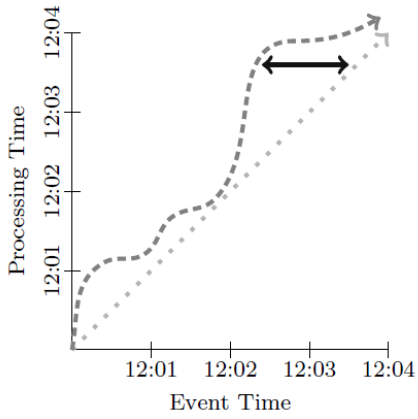


Figure 2.2: Time domain skew (Source: Akidau et al., 2015, p. 1795)

2.2 Stream processing models and systems

This section covers existing models, approaches and systems relevant to the research questions in Chapter 1. First a a stream processing model to balance correctness, latency and costs is explained. Then, windowing approaches based on data content are described before summarizing the CAESAR system and the TEF-SPARQL query-language.
Google's Data Flow Model is a stream processing model to balance correctness, latency and costs. It separates the application logic and the running engine. The model is based on four key questions (Akidau et al., 2015, p. 1793f.):

Q1 What results are calculated?

Q2 Where in event time they are being computed?

Q3 When in processing time they are materialized?

Q4 How earlier results relate to later refinements?

Q1 is about the transformations of the incoming streaming data, meaning transformation like counting, aggregate functions or data mining algorithms. Q2 describes the windowing concept (using event time), which includes traditional windows like fixed or sliding windows. But it also enables the user to define more complex windows like session windows by grouping and merging windows based on attributes and overlapping time periods. Session windows are introduced in more detail in the Section 2.2. Data Flow uses triggers for allowing the user to define when in processing time results should be emitted. They are also used to offer different options (discarding, accumulating and retracting) on how earlier results should affect later refinements (Akidau et al., 2015, p. 1797f.). Those operations are only possible since they differentiate between event and processing time and acknowledge the fact that there can exist a skew between generation of the event (event time) and processing the event by the streaming engines (processing time) (Akidau et al., 2015, p. 1794f.).

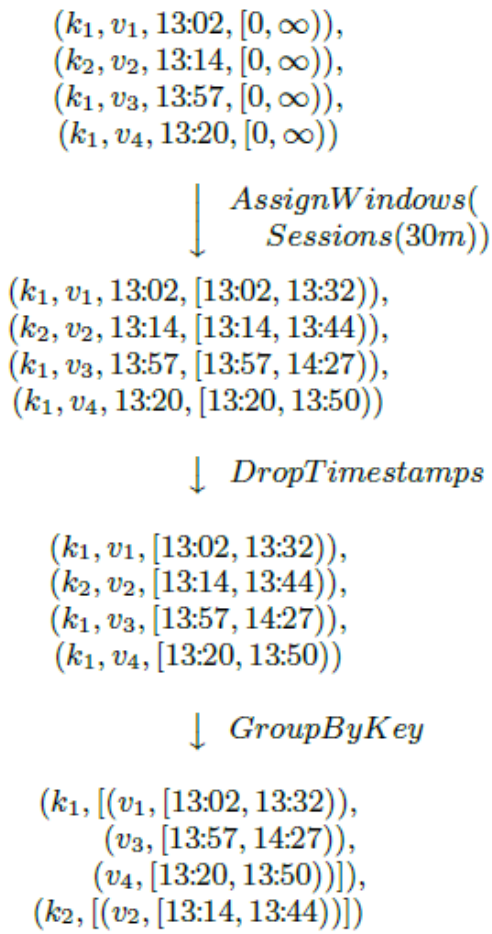


Figure 2.3: Session windows (extract) (Source: Akidau et al., 2015, p. 1796)

The Data Flow model requires a 4-tuple (key, value, event_time, window) for event-time windowing. With those tuples various windows can be constructed by `AssignWindow` and `MergeWindows` operations. The `MergeWindows` operation allows to create data-driven windows, like session windows.

The procedure of creating session windows is the following (Akidau et al., 2015, p. 1796, see Figure 2.3). The 4-tuples with a global window (from zero to infinity) enters the system, then the `AssignWindow` operation updates the window with the start time equal to the event time and the end time as start time plus session period, e.g. 30 min. Then event timestamps are dropped, the elements are grouped by a defined attribute, e.g. user id. The result of the grouping operation is used by the `MergeWindows` operation to create non-overlapping windows per key. After grouping by the key attribute the event time is updated with the end timestamp of the resulting window.

Apache Flink is an open-source system, which allows batch and stream processing in

one single engine in a distributed fashion. Carbone et al. (2015) argue that most of today's data are produced continuously and most of current data processing happens by splitting the data in arbitrary chunks and processing them in a batch mode. Therefore batch data processing is only a special case of stream processing and both can be processed by using one programming model and execution engine.

Using one programming model and executing engine for batch and stream processing has several advantages like harmonizing and transforming data only once for diverse use cases or simplify the overall IT architecture in comparison to implement the different layers of the LAMBDA architecture (Marz and Warren, 2015) with different software stacks.

Figure 2.4 shows the software stack of Apache Flink. On top of the "distributed streaming dataflow" two APIs are available: Data Set for Batch Processing and Data Stream for Stream Processing.

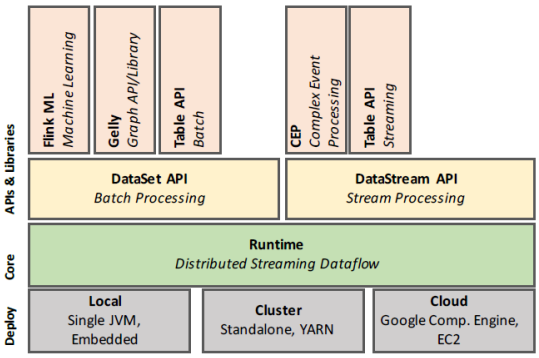


Figure 2.4: Apache Flink - software stack (Source: Carbone et al., 2015, p. 30)

Independently of the used API each Flink program is executed as a "directed acyclic graph (DAG), that consists of: (i) stateful operators and (ii) data streams that represent data produced by an operator and are available for consumption by operators" (Carbone et al., 2015, p. 30).

By using the *DataSet API* the programmer can use well-known abstractions (relations) and operators (joins, projection, aggregations) to write batch-oriented programs. It also integrates query optimization techniques suitable for queries over bounded data sets. (Carbone et al., 2015, p. 35f.) On the other hand the *DataStream API* allows the usage of sophisticated windowing operators and supports also out-of-order data processing by using watermarks based on processing-time and event-time. (Carbone et al., 2015, p. 33f.)

By transforming all batch and stream computations into a DAG, Apache Flink is able to distribute data as well as their processing. Figure 2.5 displays the processing model of Apache Flink (Carbone et al., 2015, p. 30). The transformation of the Flink program into a DAG is done by the *client*. The *JobManager* is responsible to monitor the distributed execution of the DAG. A Flink cluster can consist of multiple TaskManagers. Easy *TaskManager* can offer multiple Task slots for execution. All scheduling and monitoring

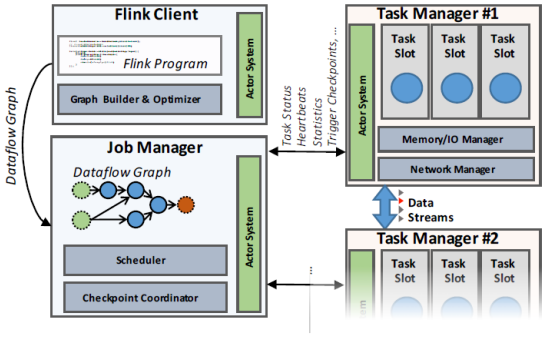


Figure 2.5: Apache Flink - process model (Source: Carbone et al., 2015, p. 30)

of the Task Managers is done by the JobManager.

Data driven windows

Session windows are one example of data-driven windows, because they require to read a key field of each single event, e.g. a user id, customer id or sensor id. Only then a session windows based on that key can be created.

Grossniklaus et al. (2016) use data-driven windows called frames. They point out that "frames are simply sub-sequences of a stream (technically the starts and ends of those subsequences)" (Grossniklaus et al., 2016, p. 13). This is similar to the resulting window timestamps of session windows, so that session windows can be compared to one type of frames, the *aggregate frames*. The difference is that Grossniklaus et al. (2016) define that aggregate frames are ending a frame, "when an aggregate of the values of a specified attribute within the frame exceeds a threshold work for this case" (Grossniklaus et al., 2016, p. 14). The end of a session window is reached when in specified time period no event for that key is generated. For example, if the time period is one minute, then the end of a session window is reached when the last detected event for a key was over one minute ago.

In session windows the value of the specified attribute is only used for merging/aggregating overlapping windows, but not adjusting the window size, like in aggregate frames. Those aggregate frames are very useful to derive specific contexts, which are this discussed in the next section. Other types of frames are threshold frames, delta frames and boundary frames.

Threshold frames are related to aggregate frames, but while aggregate frames the aggregated value of specified attribute needs to exceed a threshold, in threshold frames it is sufficient that a single value of a specified attribute of the data stream tuple falls below or exceeds a specified threshold value. Therefore, threshold frames are used to detect "interesting regions" in the data, in which further processing could be useful (Grossniklaus et al., 2016, p. 14). A new *Delta-Frame* is generated if the "value of particular attribute changes by more than an amount of x"(Grossniklaus et al., 2016, p. 14), so it detects regions of rapid change in the data. The fourth type of frames are

boundary frames, which can be used to create heat maps, e.g. soccer players' movement during a game. So the playing field is divided in different grid cells and every time an event enters the systems which crosses the borders of grid cell a new boundary frame is created.

All four types of frames produce as a result the start and end timestamps of interesting regions in the data. The framing scheme consists of local conditions (data-dependent), which define the type of frame. Global conditions (data-independent) ensure that the generated time interval has a defined minimum or maximum duration. Furthermore they are based on data content of the streams and not on external characteristic like fixed time intervals. Nevertheless the framing scheme consists of global conditions to ensure a specific length of frames, for example.

By using threshold frames it is possible to create "fixed" windows by checking the values of event attributes inside a data stream. It is not possible to adapt window deriving when a specified threshold is reached. This is possible by using data-driven, User-Defined Windows (UDF). Carbone, Traub, Katsifodimos, Haridi, and Markl (2016) provide as a motivating example for UDFs the monitoring of stock quotes. If the stock price is greater than $10, every five minutes a window, containing the weighted average price of the last 10 minutes, is created. But if the stock price falls under $ 10 dollars, the windowing should be changed with a slide of 2 minutes and a range of 5 minutes.

2.3 Context-awareness and state management

More complex global conditions (as used in Frames) could be seen as contexts, which are realized in the Context-Aware Event Stream Analytics in Real-time solution (CAESAR) by Poppe et al. (2016). Application contexts are defined as "real-world higher-order situations the duration of which is not known at their detection time and potentially unbounded" (Poppe et al., 2016, p. 415).

In the case of traffic management different application contexts (short: context) exist such as "congestion", "accident" or "clear". Each context with its name as well as context-deriving and context-processing queries is a context-type. Context-deriving queries are taking the incoming events as well as the current context into consideration to decide if a context-switch or context-termination is necessary. If the average speed of the last 25 cars is below 20 mph and the current context is "clear", the context-deriving queries could switch the context from clear to congestion. It is important to note that different context types could overlap like congestion and accident.

These context-deriving queries are conceptually similar to those queries which derive aggregate frames. In CAESAR the results of context-deriving queries are used to optimize the query load in the system. So only the context-processing queries for the contexts which are active at processing time are registered and are continuously evaluated. That means that in the contexts of "accident" and "congestion", accidents warnings and toll notifications are output events of the corresponding context-processing queries. While in the context "clear" those context-processing queries aren't running. The duration of

a context is a so called context-window (Poppe et al., 2016, p. 413-416).

In my opinion application contexts can be seen as the first-level. Frames can be used to derive contexts. On a second level session windows can be used to create data-driven windows inside a context window. The third level are facts. TEF-SPARQL is a query-language for time annotated event and fact Triple-Streams, facts "are things that are true for specific amount of time" (Kietz et al., 2013, p. 9). The provided example for a fact is the event "Peter owns a book", since it is true after he bought the book, but could be invalidated after he looses it or sells it again. Another fact could be: "Maria is watching the online tv channel XYZ". It can be used to construct and update temporal facts out of event triples (in this case RDF triples) and combine them with time annotated event streams (Kietz et al., 2013). With TEF-SPARQL the generation of facts and querying the event stream is combined which could have a negative impact on the performance and doesn't allow to share facts across different event streams, at least in my opinion. By considering a external storage for facts bi-temporal databases important to consider. Bi-temporal database differentiate between different notions of time like valid/application time and transaction/system time. They also offer special operations for handling those different notions of time. (Snodgrass, 2000; Kaufmann, Fischer, May, and Kossmann, 2014)

Margara et al. (2017) use a concept similar to facts which they name "state". Furthermore, their proposed model consists of three different components: state management, stream processing and state. They separate the management (including updates) of the states or facts from the stream processing and provide the possibility to save specific states permanently. Therefore it could be possible to weight different states according to the application context.

In the remaining chapters of this thesis the following wording applies: The highest level is the application context (as defined in CAESAR). Data-driven windows (e.g. frames) are used by context-deriving queries. Context-processing queries can be used to derive facts, which are valid during a specific period of time in a context window or can be also saved permanently to make them available for context-processing queries of other contexts. States as used by Margara et al. (2017) will not be used, because they are used extensively by the database community (stateful operators), but conceptually they will remain as overall context information and context-specific information.

3

Conceptual model

This section presents the conceptual model of Context-aware, Fact and Background integrated dynamic Stream Processing (CoFaBidSP). It first presents details about two case studies in the field of smart cities (traffic management) and entertainment (soccer monitoring). Next, it describes the components and data processing flow of that model.

3.1 Case studies

I motivate this study through two case studies: traffic management and soccer monitoring. In the field of traffic management there are several data sets like the Linear Road Benchmark (Arasu et al., 2004) or the Vehicle Traffic dataset of City Bench (Ali, Gao, and Mileo, 2015). In this thesis I refer to the data provided by Brőring et al. (2015), because it has several advantages. First, it contains real world data. Second, the data in this data set was collected with a modern hard- and software architecture. Third, important data points were selected and stored in JSON (see Listing 3.1) or RDF format. Well-defined APIs and a data model were used for this purpose (enviroCar, 2015).

Listing 3.1: Excerpt of an event of the traffic management case study

```
{"features":
[{"type":"Feature",
 "geometry":{"type":"Point",
   "coordinates":[6.4847174678758375,51.22546715521443]},
 "properties":
  {"id":"579634f9e4b086b281bf935e","time":"2012-01-01T00:06:
     44Z",
   "sensor":
                          {
                           "type":"car",
                           "properties":{"engineDisplacement":
                              2200,"model":"Vectra C Caravan",
                     "id":"5750591ee4b09078f98673d8","fuelType
                        ":"gasoline",
```

```
12                      "constructionYear":2004,"manufacturer":"
                           Opel"}
13                           }
14 ...]}
```

The data set consists of several entities with information about sensors, tracks and measurements. One event (record) consists of the following attributes: measurementID, sensorID, trackID, longitude, latitude, timestamp, speed, consumption, rpm, road segment. Road segment is an artificial attribute, which is not included in the original data set, to simplify the explanation. The data is collected by private citizens, who equipped their private cars with a sensor. To collect measurements of their tracks the citizens were provided an app to upload the recorded events to the enviroCar server (every five seconds is an measurement during a track recorded).

The second case study considers the measurements of a football match, recorded by collecting data from sensors in the ball, in the shoes of the soccer players, the referees and the gloves of the goal keepers (Mutschler, Ziekow, and Jerzak, 2013). Each event has the following schema: an unique id (sid), timestamp in picoseconds (ts), sensor coordinates (x,y,z), velocity (v), acceleration (a), direction vector (vx, vy, vz), acceleration vector (ax, ay, az). The coordinates (0,0,0) represent the middle of the football field.

3.2 Data model: Events, Context, Facts and Background Knowledge

The proposed conceptual model uses various types of information, see Figure 3.1. One or more continuous data streams serve as input for the context-aware distributed event processing system.

The output is sent to a sink, e.g. an external database. Application contexts are defined as "real-world higher-order situations the duration of which is not known at their detection time and potentially unbounded" (Poppe et al., 2016, p. 415).

Context information are used at processing time to control and change the handling of the input data streams. In the traffic management case study possible contexts could be "Day" or "Night". Additional attributes are start and end of such a context, the number of events inside a window while this context is active. In Figure 3.2 are also examples for the soccer monitoring use case shown. ts1 is the time stamp with the value 10,753,295,594,424,116 (start of the first half), ts2 is 12,557,295,594,424,116 (end of first half). The second half starts at ts3 with value 13,086,639,146,403,495 and ends at t4 with value 14,879,639,146,403,495.

In both use cases the context information is defined by domain experts. The model does not restrict the number of contexts, but only one context can be active at each point in time. It is important to mention that context information itself is not part of the distributed stream processing system. The advantages and disadvantages of this design decision are explained in Section 4.2.1.

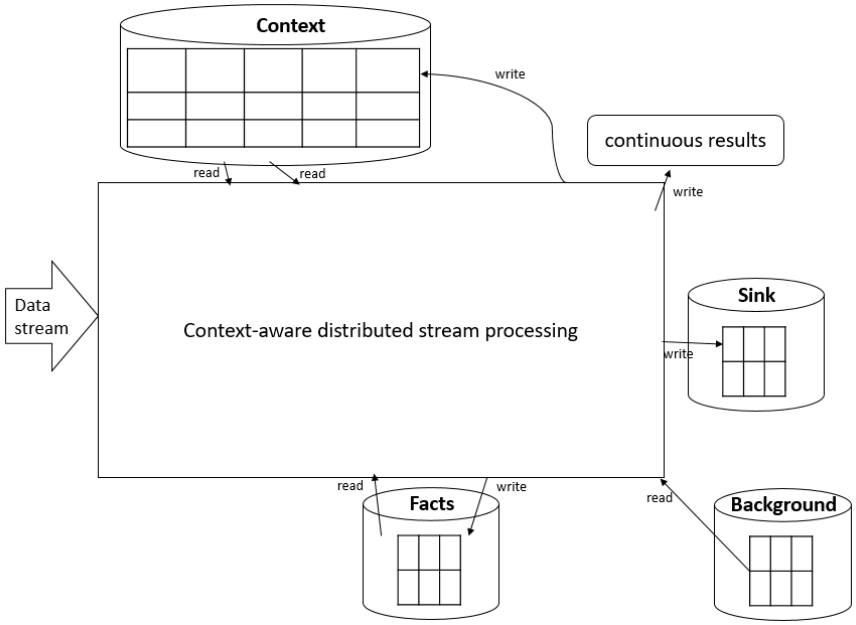


Figure 3.1: Information types (Source: Own figure)

Facts "are things that are true for specific amount of time" (Kietz et al., 2013, p. 9) Facts, however, are frequently updated by the stream processing system, since they are created or updated based on single events or aggregations inside a global windows.

Traffic management		
Name	Value	System period
Car1InRoadSegment	2	2018-01-20 14:12:23,
Car1InRoadSegment	1	2018-01-20 14:09:55, 2018-01-20 14:12:22
Car1AvgSpeed	45	2018-01-20 14:12:23,
Car1AvgSpeed	23	2018-01-20 14:09:55, 2018-01-20 14:12:22
VehiclesRoadSegment1	10	2018-01-20 14:12:23,
VehiclesRoadSegment1	20	2018-01-20 14:10:11, 2018-01-20 14:12:22
VehiclesRoadSegment1	8	2018-01-20 14:09:55, 2018-01-20 14:10:10
RoadSegment1Status	clear	2018-01-20 14:09:55,

Table 3.1: Facts of traffic management use case (Source: Own representation)

Examples of Facts are the number of elements which are currently inside processing engines or the distribution or average of an attribute of the event streams. More concrete facts in traffic management are the numbers of cars in one specific road segment, the average speed of all cars, the segments in which traffic is congested or clear (see Table 3.1). In the soccer monitoring case study, facts are the ball possession, number of shots

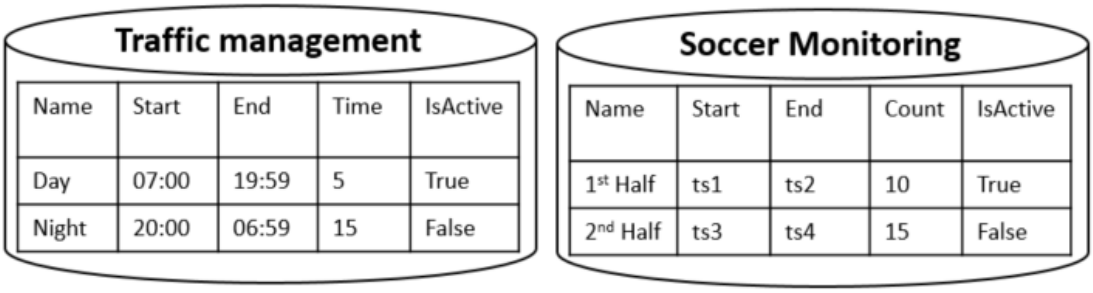


Figure 3.2: Context information (Source: Own figure)

to goal and run time statistics for a player.

To produce continuous results the stream processing system can also take into account background knowledge (e.g. from domain experts) or historical data about entities represented in the current data stream, context or facts. Examples for such background knowledge or information in the traffic management use case are the speed limit of various road segments, the mapping of geo-coordinates to road segments, holidays, historical statistical data like average count or average speed in road segments. In the soccer monitoring case study past performances of each football player, previous results between the teams, turnover of the teams or size of fan base is background information.

This model also enables further batch oriented analytics by combining the transformed data stream, versions of facts and background information, but this is out of scope of this thesis.

3.3 Processing model

This section describes the components and process flow of the Context-aware, Fact and Background integrated dynamic Stream Processing (CoFaBidSP) model.

3.3.1 Source function

After explaining the different information types of the proposed model, I describe the components to build the context-aware distributed stream processing system (see Figure 3.3) using those information types.

The “Source function” gets the data stream in the raw format and transforms it in objects which can be handled by the following components, so it is essentially a wrapper transforming the input event into the desired format for the following operations. Optionally, additional attributes can be added, like road traffic segments based on geo-coordinates. The event-stream is then copied and transferred to the two subsequent components, which run in parallel and independently from each other.

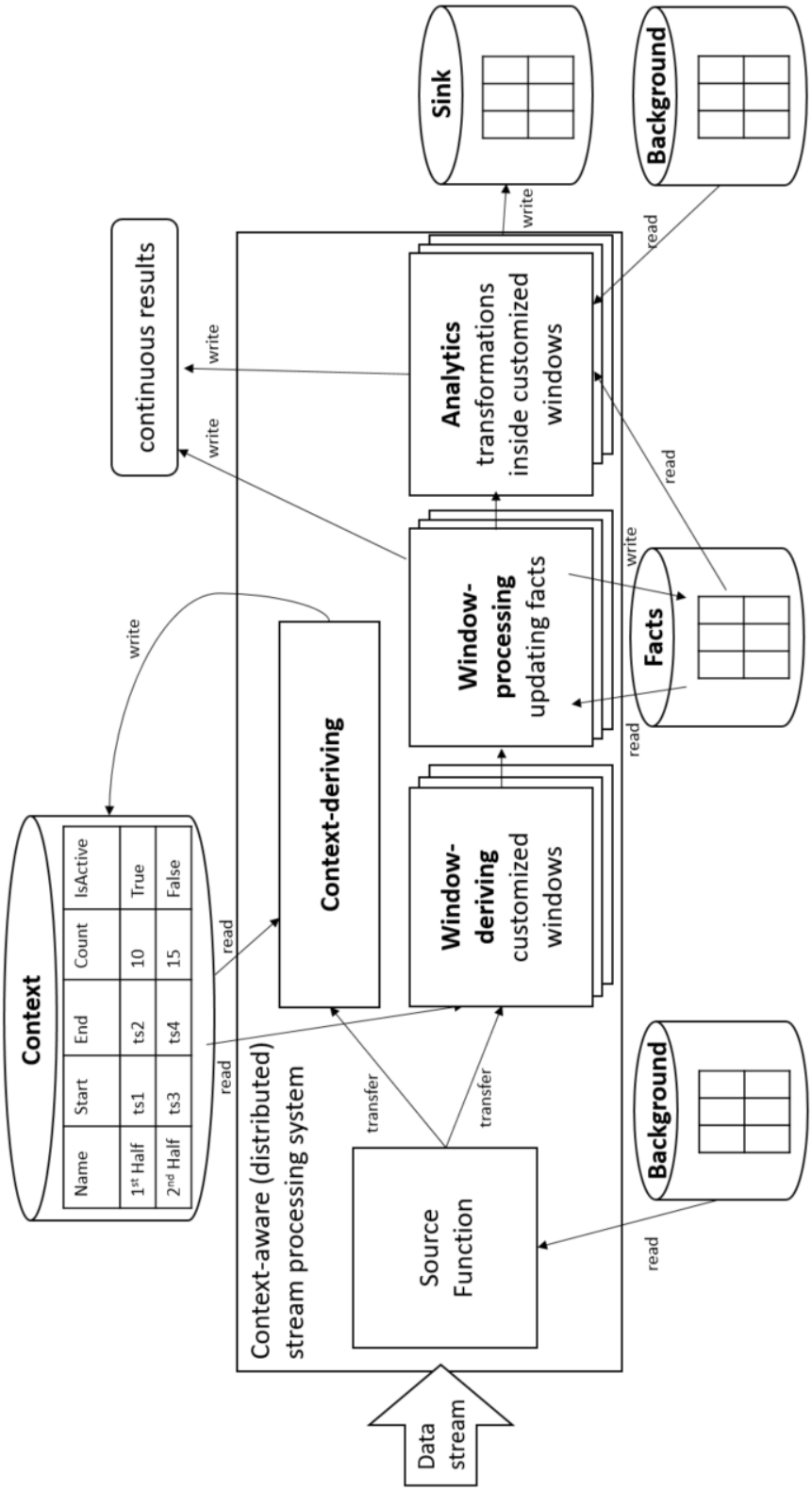


Figure 3.3: Model overview (Source: Own figure)

3.3.2 Context-deriving

While continuously receiving new events, the context-deriving component uses the attribute values of those events as well as the static context information to check if the current context is still valid or if a context switch is necessary. If such a switch is necessary, the "IsActive" Flag is updated.

For the traffic management and the soccer monitoring case study, it is assumed that all events are generated in the same time zone, contain an time stamp which can be used as an event time stamp. It is further assumed that the lag between the event time stamp and the time stamp of processing the event in the "SourceFunction" is insignificant.

3.3.3 Window-deriving

Depending on the current context the segmentation of the event stream differs. That means windows are not always constructed in the same way. One way to do this to split the incoming data stream on one or more key attributes, that would happen even before the creation of windows. It allows a logical as well as physical distribution of succeeding processing. Examples are geo-coordinates, road segment, car or for the soccer monitoring player, derived team. Consequentially further processing only has access to events containing a single value of the key attributes.

For the soccer monitoring case study, Figure 3.2 shows that the number of events differ between the first and second half. If the context "1st half" is active a count-based window of size 10 is used, otherwise a count based window of the size of 15 is used. Figure 3.4 shows the three states of the window-deriving component in the 1st half. The left one displays all incoming events delivered by the Source Function. The middle on collecting those events inside a global window. After reaching the count size of 10, the window-deriving component emits those 10 events for further processing.

Then the Context-deriving component executes a context-switch. Figure 3.5 shows the three states of the dynamically adjusted window-deriving. The left and middle parts remain the same. However the window-deriving components emits windows containing always 15 events based on the current context, which is now the 2nd half.

In general the models also allows to the have several concurrent window-deriving processes to be able to satisfy more sophisticated uses cases.

3.3.4 Window-processing

The *Window-processing* component is using events inside a customized window for calculation of additional key figures. Those key figures represent often facts. Depending on the content of the windows different facts can be calculated.

Since facts are not part of the distributed data stream processing engine, it is necessary take care of possible concurrency issues. One option to avoid concurrency issues is to use no sliding windows. If the input data stream has high velocity, it is also likely that corresponding facts are updated very often, which will results in many version of the same fact, the challenges related to this are discussed in the next section.

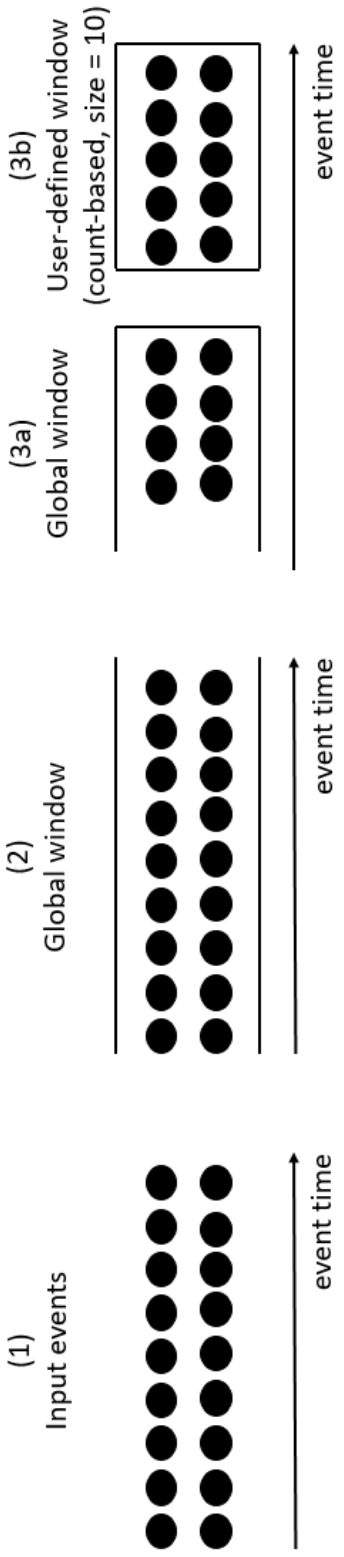


Figure 3.4: Window deriving in context of 1st half with count-based window of size = 10 (Source: Own figure)

Figure 3.5: Window deriving in context of 2nd half with count-based window of size = 15 (Source: Own figure)

The generated facts of the window-processing components are often already very valuable for getting real time insights e.g. about the traffic flow or which soccer team is dominating the game. So the model is designed such that those facts and the corresponding updates are emitted continuously.

3.3.5 Analytics

The analytics component can output results in various formats. It can dump the previously generated window into a data sink. From this intermediate storage, the data undergoes further processing, such as logging or monitoring. This is especially useful for batch-oriented analytics. Another option is to analyze the streams directly without an intermediate storage solution.

While window-processing focuses on updating facts based on the event stream, window-analytics generates complex insights by combining events, facts and background information, either in a streaming or batch-oriented fashion. In the streaming fashion, the facts and background information are converted into a data stream and are then combined with the event stream in to one, which can be used as an input for detecting complex events detection. The main challenge is to construct one event out of three different data sources, due to difference in structure, volume, data quality and (time) granularity. Otherwise it is not possible to construct patterns which can be found in the data stream. The combination is only done by using SQL-like statements including projects, aggregation, and set-operations like join and union. That means, facts and background are conceptually global windows which are combined with every generated window of previous components. The relationship between event attribute and facts is at least one-to-many, so for one event attribute, e.g. for each sensor or player more than one fact is generated with different validity periods.

By doing analytics in a batch-oriented fashion, it is assumed that the data on which queries are executed is not changing over time. Then it can be differentiated between analytics concerning only the content of a specific single window or a set of windows.

By using the data sink, which contains the content of all or set of generated windows, the model is also enabling applications using machine learning/data mining methods or graph analysis.

4

System design and implementation

This chapter explains the technical system design and provide details about the implementation of the components. The next section specifies requirements, which the implementation should satisfy.

4.1 Requirements

I identified the following requirements based on the conceptual model:

R1 **Distributed processing of data stream events**
The data streaming system should process incoming events with a centralized as well as distributed fashion. In a centralized fashion a single as well as parallel processing on one computer should be possible. In a distributed fashion, independent computers should be able to process events in parallel. For this scenario, a component should take care of scheduling and monitoring of these independent computers.

R2 **Support for reading and updating context-information inside a streaming data pipeline**
The implemented system must read and write externally stored context information to control and adapt the computations inside a data streaming pipeline.

R3 **Support for reading and updating facts inside a streaming data pipeline**
3.1) Inside a data streaming pipeline the system should read and write facts into an external component. This external component must be transaction-safe (atomic, consistent, isolated and durable) (Tanenbaum and Van Steen, 2007, p. 21).
3.2) Furthermore the external component should provide support for at least one notion of time such as valid/application time, transaction/system time or bitemporal time. (Kaufmann et al., 2014; Snodgrass, 2000)

R4 **Provision of different customized windows based on current contexts**
The implemented system must adapt the deriving of windows based on current contexts, without stopping or restarting the overall data streaming pipeline.

R5 **Allow for context-switches in a continuous data processing pipeline**
The system should switch contexts based on predefined attribute values of incoming events (e.g. event time reaches a specified end time). Furthermore no restrictions for the number of context switches should apply.

R6 **Combination of events, facts and background information in continuous queries**
The system should enable building continuous queries combining events, facts and background information. The support for ad-hoc queries is also desirable.

4.2 Components and their functions

In this section I explain the technical system design and provide implementation details of the components. The architecture consists of two main parts: (1) a PostgreSQL database with an extension to support temporal tables, (2) Apache Flink, an Open Source Stream Processing Framework of the Apache Software Foundations.

PostgreSQL is a relational database to implement the external component responsible to storing context, facts and background information. An extension for temporal tables allows to implement the requirements 3.1) and 3.2). In a PostgreSQL database, named "soccer_monitoring", three tables are used to store the context, facts and background information. The data streaming pipeline executed in Apache Flink is programmed in Java.

4.2.1 Context, Facts and Background Information

Context and background information

Both case studies introduced in Section 3.1 uses context information predefined by domain experts. This context information is centralized and not subject to frequent updates, except the flag which context is active. Figure 4.1 shows the content of the context table used in the system.

```
soccer_monitoring=# select * from context;
 id |    name     |       start       |      endtime      | windowsize | isactive
----+-------------+-------------------+-------------------+------------+----------
  1 | first half  | 11016510166923100 | 11023063135026300 |      15444 | f
  2 | second half | 11023063135026301 | 11027991098119800 |      17333 | t
```

Figure 4.1: Implementation of context information in PostgreSQL (Source: Own figure)

The stream processing system has read access for all fields, but can only update the column "IsActive". In contrast, all the fields except "IsActive" can be only updated by domain experts. However, as mentioned before even if this model allows updates, it is not designed for frequent updates. Contexts are non-overlapping and at each point in time, exactly one context is active.

Background information is static and it is expected to have high data quality. Background information is stored in a relation centralized database by default, but due to its nature it is also possible to load it into memory and replicate it for distributed computing.

Facts

Facts are frequently updated by the stream processing system, since they are created or updated based on single events or aggregations inside a global window. This global window can be keyed (e.g. by geo-coordinates, segments, sensorID, playerID) or non-keyed. If a key is provided, a distributed execution per key is possible. Facts are stored in a temporal database. That means that every update of an record is automatically recorded with the corresponding system time stamp. To avoid synchronization issues this database is centralized and transaction safe.

```
soccer_monitoring=# select * from facts;
 Firefox Web Browser          | count | value | unit  |          created_at           |             sys_period
------+----------------------+-------+-------+-------+-------------------------------+-----------------------------------
  5 | team2ShotsOnGoal       |     5 |     0 | shots | 2018-03-14 13:57:21.064237+01 | ["2018-03-14 13:57:21.064237+01",)
  1 | player1RunningDistance |     0 |  5234 | m     | 2018-03-14 13:56:22.048512+01 | ["2018-03-14 14:01:15.031569+01",)
  3 | player2RunningDistance |     0 |   947 | m     | 2018-03-14 13:55:37.057423+01 | ["2018-03-14 14:02:13.767703+01",)
  4 | team1ShotsOnGoal       |     3 |     0 | shots | 2018-03-14 13:57:04.632582+01 | ["2018-03-14 14:03:38.831993+01",)
(4 rows)

soccer_monitoring=# select * from facts_history;
 id |          name          | count | value | unit  |          created_at           |                            sys_period
----+------------------------+-------+-------+-------+-------------------------------+-------------------------------------------------------------------
  1 | player1RunningDistance |     0 |  5000 | m     | 2018-03-14 13:56:22.048512+01 | ["2018-03-14 13:56:22.048512+01","2018-03-14 14:01:15.031569+01")
  3 | player2RunningDistance |     0 |   233 | m     | 2018-03-14 13:55:37.057423+01 | ["2018-03-14 13:55:37.057423+01","2018-03-14 14:01:45.560165+01")
  3 | player2RunningDistance |     0 |   826 | m     | 2018-03-14 13:55:37.057423+01 | ["2018-03-14 14:01:45.560165+01","2018-03-14 14:02:13.767703+01")
  4 | team1ShotsOnGoal       |     2 |     0 | shots | 2018-03-14 13:57:04.632582+01 | ["2018-03-14 13:57:04.632582+01","2018-03-14 14:03:38.831993+01")
(4 rows)
```

Figure 4.2: Implementation of Facts in PostgreSQL (Source: Own figure)

Figure 4.2 shows the implementation of Facts for the soccer monitoring case study. The columns `created_at` and `sys_period` are filled by the temporal database extension automatically. The column `created_at` contains the system time, when this record was created. In both tables (`facts` and `facts_history`) the column `sys_period` has always a time stamp interval. In the `facts` table only the current record is displayed, containing one time stamp in the column `sys_period`. This time stamp represents the system time of the last update. The end time stamp is not filled, which means this record is currently valid.

The `facts_history` table contains the whole history for each fact. As shown in Figure 4.2, for all records the column `sys_period` has two time stamps, representing the transaction time period, in which the value/count of a fact was valid. It is important to mention that these time stamps represent the processing time, on which the value of the facts was calculated. They are not based on the event time.

4.2.2 Stream processing system

Apache Flink is used to implement the components of the context-aware distributed stream processing system of the conceptual model, introduced in Figure 3.3. Each component of the conceptual model is realized as one Java class. In the current prototype, the

Source Function, Context-deriving and Window-deriving are implemented entirely. For the Window-processing and Analytics components only a basic implementation exists.

All implementations are developed and tested by using the data of the soccer monitoring case study. The Source function as well as the Context-deriving components apply the event time in nanoseconds (the event time stamp provided in the soccer monitoring data are in picoseconds). The Window-deriving component is using count-fixed windows and does not take into account the event time.

Source function

Figure 4.3 shows the raw data of the soccer monitoring case study. Each event has an unique id (sid), timestamp in picoseconds (ts), sensor coordinates (x,y,z), velocity (v), acceleration (a), direction vector (vx, vy, vz), acceleration vector (ax, ay, az). The attributes are separated by a comma.

```
8,11027988538138665,-6808,-10220,231,235485,3089177,3016,2419,9222,-1378,5847,7994
4,11027988648803696,1224,9860,16,3803822,15872221,965,-9951,182,7113,-6034,3604
10,11027988696704673,-5922,14824,612,196196,2655471,1074,3216,-9407,-2283,9040,-3613
38,11027988740358224,227,10756,540,5789200,9540327,-1715,-9825,725,4282,8953,1222
62,11027988743603763,21528,28465,235,303042,2707815,8168,-5676,1021,7867,-5721,-2315
14,11027988778429561,22860,-21280,-27,3158931,6234620,-7555,-6524,-580,-6437,-6425,-4155
44,11027988861500306,31807,-321,-134,2185967,16749290,-6894,-7065,1597,-7252,-6856,630
54,11027988995873357,9452,-1614,172,2437388,16934908,-6283,-7505,-2045,8384,5388,-811
```

Figure 4.3: Examples of the soccer monitoring data (Source: Own figure)

The SoccerEvent Java class maps this input to a Java Object representing each event of the source data. Furthermore, it maps the sensor ID to a player ID, since each player has two sensors. For the goal keeper, it maps all four sensors (hands and feet) to the corresponding ID. The Source Function is implemented by extending the `RichSourceFunction` of the Apache Flink Framework to be able to use the event time for the following processing, show in Figure 4.4.

```
SoccerEvent soccerEventOutput = SoccerEvent.fromString(line);
sourceContext.collectWithTimestamp(soccerEventOutput, getEventTime(soccerEventOutput));
```

Figure 4.4: Implementation Source Function (Source: Own figure)

The implementation of this Source Function is used by the components for Context deriving and Window deriving, which are explained in the following sections. For the traffic management and the soccer monitoring case studies all events are generated in the same time zone.

Context deriving

The Context deriving components are implemented as a Java program. While continuously receiving new events from the Source function, the Java program uses the Apache Flink Framework to create tumbling windows based on event time. The `ProcessAllWindowFunction`

checks for every event whether the current context is still valid. That means that if the time stamp of an event is not between the start and end time stamps of the current context, a context-switch is done.

The switch is realized by setting the current context to `isactive=false` and the new context to `isactive=true`. The Apache Flink Framework has to be able to serialize all Java objects. Therefore it is important to handle PostgreSQL connection in the `open` and `close` method of the `ProcessAllWindowFunction`.

Window-deriving

Depending on the current context, the definition of the created windows should differ. That means that the implementation should allow user-defined windows, as described in Section 2.2. One way to do this is to split the incoming data stream by one or more key attributes. Examples for such attributes are geo-coordinates, roadsegmentID, carID for the traffic management use case or playerID or teamID for the soccer monitoring. Consequentially, further processing can only access the events containing a single value of the key attributes.

The detailed process is as follows: The events are collected in a non-keyed global window. The *CustomCountTrigger* extends the Apache Flink default count trigger and has two processing steps. First, at window creation time a database connection is opened to retrieve which context is active and saves the corresponding information in memory. The second method `onElement` is called for every arriving event in this window. It verifies if a window should stay open or should be closed. If the window is to be closed, the window events are available for further processing.

In the soccer monitoring case study, the current context defines that each window contains 17'333 events (see Figure **??**). In case an internal event counter in the `invoke` method reaches 17'333, the window closes and emits the last 17'333 events. Those windows are conceptually related to Aggregate Frames by Grossniklaus et al. (2016). Another option would be to use different time sizes of tumbling windows. In the traffic management use cases with non-keyed global window, the trigger function closes the window after the 5 minutes since the last event was added to the window, during night it is closed after 15 minutes. That would mean that each window with active context "Day" consists of all the events of the last 5 minutes based on the event time.

In general the architecture allows for the possibility to have several concurrent window-deriving processes to have non-keyed and keyed customized windows at the same time. For formal definitions of user-defined windows see Carbone et al. (2016).

Window-processing

The *Window-processing* component is also implemented in Java using the Apache Flink Framework. It takes the events inside a customized window for calculation of various additional key figures. Those additional key figures are facts and therefore are saved in the `facts` table of the temporal database, as shown in Figure 4.2. Essentially every

window updates the facts. The storing of previous results for this factID is saved in the `facts_history` table using database triggers.

Since facts are not part of the distributed data stream processing engine, it is necessary take care of possible concurrency issues. One option to avoid concurrency issues is to use no sliding windows. Another option is that every update inside window-processing is directly followed by commit. Furthermore assuming that every update statement is only acquiring a row lock the execution time is very fast. By using active database trigger, every update saves the previous version of the record. If the input data stream comes with a high frequency, it is likely that corresponding facts are updated very often, which may result in many versions of the same fact. The challenges to cope with that are discussed in the next section.

The generated facts of the window-processing components are often already very valuable for getting real-time insights about how the traffic flow is going or which soccer team is dominating the game. This means the model is designed in a way that those facts and the corresponding updates are emitted continuously.

Apache Flink offers various functions for calculating facts. Examples are the Reduce, Aggregate or Fold functions. The official Apache Flink documentation provides further details for those functions. It is crucial to mention, that the API is not stable and may change between releases, e.g. the Fold Function can be used in Flink Version 1.3 but is deprecated in version 1.4.

Analytics

The analytics component can have three different outputs. The easiest one would be, that previously generated window is dumped into a data sink, which can be used for further processing, logging or monitoring. The second possible output is to do analytics on streams. The last option is to do batch-oriented analytics.

While window-processing focuses on updating facts based on the event stream, window-analytics is responsible for generating complex insights by combining events, facts and background information, either in a more streaming or batch-oriented fashion. In the streaming fashion the facts and background information are converted into a data stream. The `join` operations of Apache Flink allows to get one stream with three different sources.

The main challenge is to construct one event out of three different data sources, due to difference in structure, volume, data quality and (time) granularity. Otherwise it is not possible to construct patterns which can be found in the data stream. The combination is only done by using SQL-like statements including projects, aggregation, and set-operations like join and union.

For using SQL statements it is necessary to use the Table and SQL API. A data stream can be registered as a table. This table is used by SQL statements, see Figure 4.5. For the detection of Complex Event Patters Apache Flink offers a dedicated library.

Facts and background are conceptually global windows which are combined with every generated window of previous components. The relationship between event attributes and facts at least one-to-many, so for one event attribute e.g. sensorID, roadSegmentID,

```
// register the DataStream as table
tableEnv.registerDataStream( name: "soccerEvents", soccerevents);
Table table = tableEnv.sqlQuery("SELECT DISTINCT sensorId FROM soccerEvents");
```

Figure 4.5: data stream as table (Source: Own figure)

playerID more than one fact is generated with different validity periods. To perform a semantically correct join it is proposed to use the event timestamps of the event to access the "correct" fact, acknowledging, that this could be an "outdated" fact. The same also applies to background information.

By doing analytics in a more batch-oriented manner, it is assumed that the data on which queries are executed is not changing over time. It can then be differentiated between analytics concerning only the content of a specific single window or a set of windows. Apache Flink offers libraries for machine learning or graph analysis.

By using the data sink, which contains the content of all or set of generated windows, the model is also enabling to use tools outside of the Apache Flink Framework.

4.3 Summary

The previous sections described the system architecture and explained implementation details of the current prototype and possible extensions to satisfy the defined requirements of Section 4.1. R1 is satisfied implicitly by using Apache Flink, a distributed streaming platform. R2 and R3a) are satisfied by using extending specialized process function to read and write to PostgreSQL data base inside a Apache Flink program. By using a temporal database extension for PostgreSQL support for transaction/system time is realized, which relates to R3b). R4 and R5 are also satisfied by implementing customized triggers in Apache Flink to create User-defined windows based on the current context, see Section 4.2.2. The realization of R6 is also shown in a basic implementation in Section 4.2.2.

5

Evaluation

In this chapter I present the evaluation results for the implemented prototype. First, the setup environment is introduced. Section 5.2 describes the results of quantitative analysis. The experiments use the data set of the soccer monitoring case study and compares the effects of parameter changes (such as context switches) on the metrics run time and events per second. The last section compares the functionality of the implemented system to other available streaming solutions.

5.1 Setup Environment

The experiments were conducted on a Lenovo Thinkpad T560 with an Intel Core i7 processor (2 physical, 4 logical cores) with 16 GB RAM inside a Virtual Box VM running Ubuntu 16.04 LTS. The VM used one core and 10 GB RAM. Java was installed in version 1.8, Apache Flink in Version 1.4. The version of PostgreSQL is 9.5.10, the temporal table extension is used in version 1.2.0. For the JobManager one GB memory was assigned, the JobManager could use up to six GB memory.

5.2 Quantitative Analysis

The quantitative analysis is done by using various data subsets of the soccer monitoring case study to investigate the prototype performance. The components Context-deriving and Window-deriving are tested separately, since the first one is using event time while the second one is based on processing time. Table 5.1 shows details about the metrics used in the quantitative analysis. All experiments are conducted on a single computer with one JobManager und one TaskManager (with two Task slots).

Those metrics are measured by using a parameter space of size 4 as shown in Table 5.2. The applied methodology follows three principles: First, the system runs in a default configuration of all parameters to set a baseline. Second, only one parameter is changed, while all other parameters remain unchanged. Three, every parameter configuration runs three times and the average value of this three runs is used for the visualizations.

The soccer monitoring case study consists in total of 49'576'080 input events. Due to memory restrictions only a subset of the whole data set is used in the experiments,

Metric	Calculation
Events per second	Number of events / processing time,
Run time in seconds	processing end time - processing start time

Table 5.1: Overview metrics (Source: Own representation)

ID	Parameter	Note
1	Number of events	
2	Number of contexts	
3	Number of context switches	
4	Window Count Size	Window Count Size for Window deriving

Table 5.2: Parameter space (Source: Own representation)

represented by parameter one, the number of events. The timestamps of the contexts were adjusted accordingly.

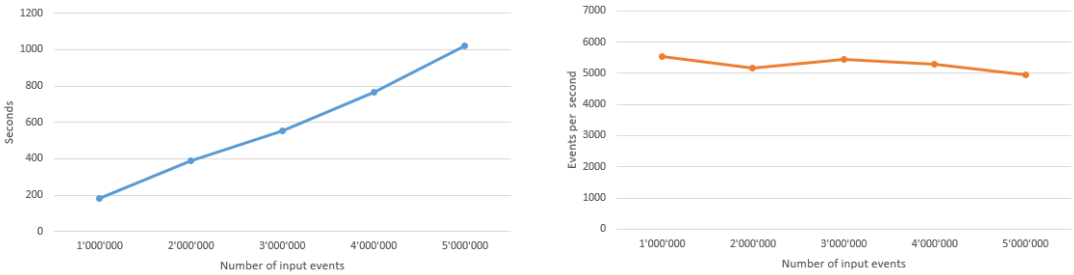
5.2.1 Number of events

This section provides insights about the effects of changes in the parameter "Number of events". All other parameters are hold constant as shown in Table 5.3.

ID	Parameter	Values
1	Number of events	1'000'000; 2'000'000;3'000'000;3'000'000;5'000'000
2	Number of contexts	1
3	Number of context switches	0
4	Window Count Size	30000

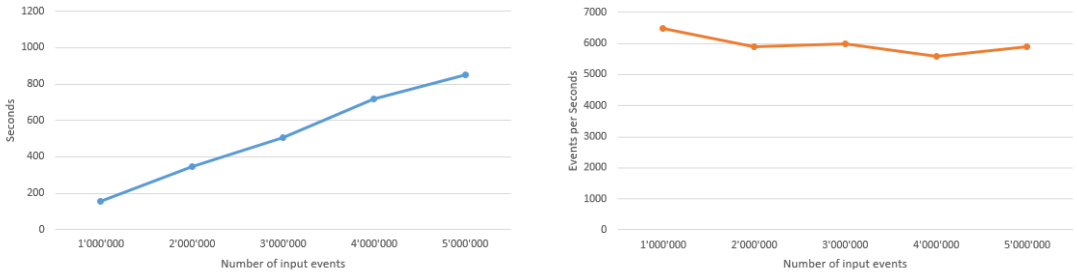
Table 5.3: Number of events - Parameter space (Source: Own representation)

Figure 5.1a displays the measurements of the run time metric by increasing the input events. It shows that the run time increases linear with the number of input events. Figure 5.1b shows that the metric "events per second" does not change drastically by increasing the number of events.
Figure 5.2a and Figure 5.2b display the measurements of run time and events per second for the component window deriving. The run time increases linear and the events per second metric is not changing drastically.



(a) run time (Source: Own figure) (b) events per second (Source: Own figure)

Figure 5.1: Number of input events - Context deriving measurements (1 context)



(a) run time (Source: Own figure) (b) events per second (Source: Own figure)

Figure 5.2: Number of inputs events - Window deriving measurements (1 context)

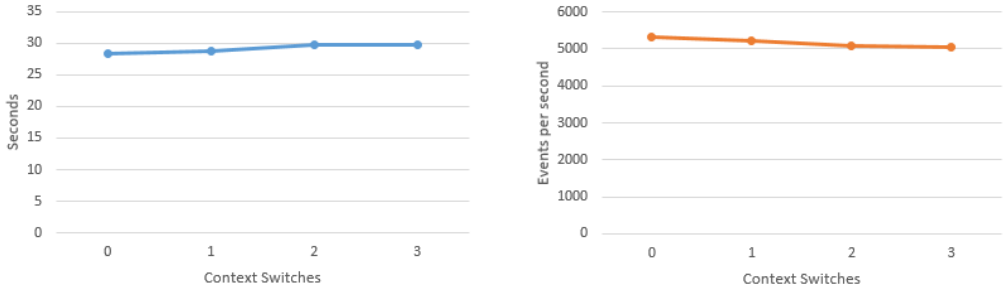
5.2.2 Number of contexts and context switches

This section provides insights about the effects of changes in the parameters “Number of contexts” and “Number of context switches”. All other parameters are hold constant as shown in 5.4.

ID	Parameter	Value range
1	Number of events	150’000
2	Number of contexts	1 - 4
3	Number of context switches	0 - 3
4	Window Count Size	30000

Table 5.4: Context switches - Parameter space (Source: Own representation)

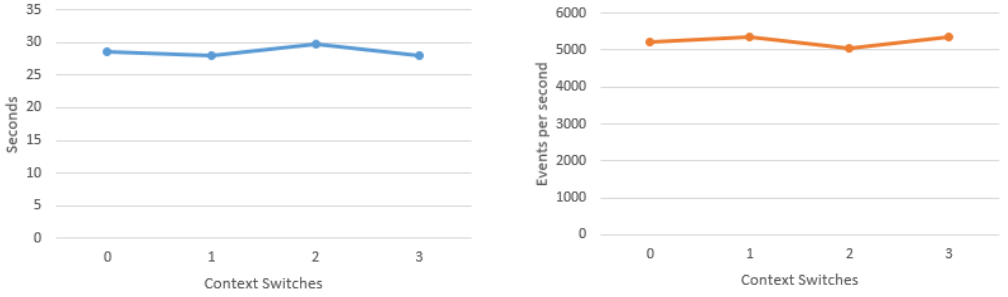
Figure 5.3a displays the measurements of the run time metric by increasing the number of contexts as well as the context switches while holding the other parameters constant. It shows that the run time does not change with an increase of context switches. Figure 5.3b shows that the metric “events per second” does not change drastically by increasing the number of contexts and context switches.
Figure 5.4a and Figure 5.4b display the measurements of run time and events per second



(a) Run time (Source: Own figure) (b) Events per second (Source: Own figure)

Figure 5.3: Context switches - Context deriving measurements with 150 000 input events

for the component window deriving. The run time and the events per second metrics are not changing drastically by increasing the number of contexts and context switches.



(a) run time (Source: Own figure) (b) Events per second (Source: Own figure)

Figure 5.4: Context switches - Window deriving measurements with 150 000 input events

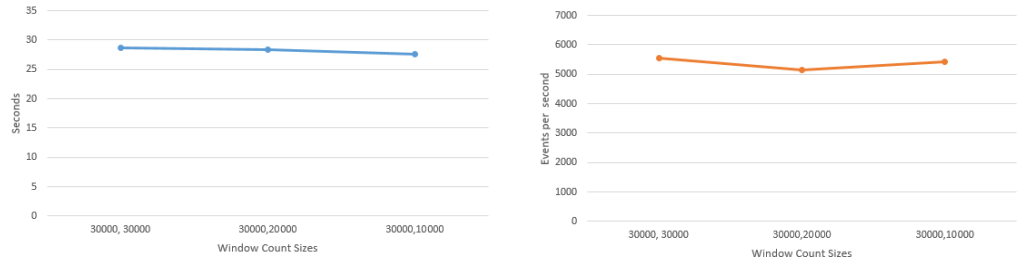
5.2.3 Window count size

This sections provides insights about the effects of changes in the parameter "Window Count Size". All other parameters are hold constant as shown in Table 5.5.

ID	Parameter	Values
1	Number of events	150'000
2	Number of contexts	2
3	Number of context switches	1
4	Window Count Size	30000,30000; 30000,20000; 30000,10000

Table 5.5: Window Count Size - Parameter space (Source: Own representation)

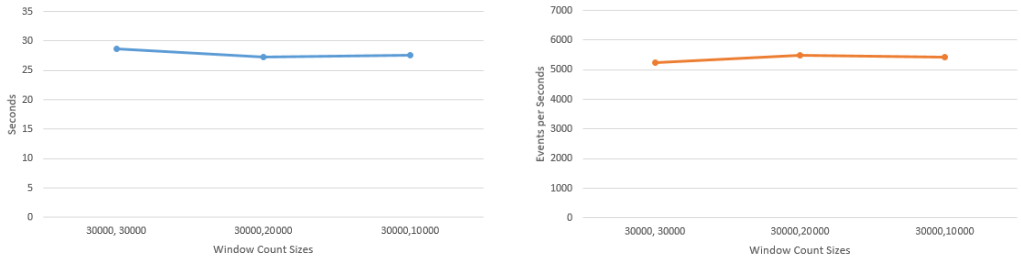
Figure 5.5a displays the measurements of the run time metric by decreasing the window count size of the windows created in the second context. It shows that the run time does not change. Figure 5.5b shows that the metric "events per second" does not change drastically.



(a) Run time (Source: Own figure) (b) Events per second (Source: Own figure)

Figure 5.5: Window count size - Context deriving measurements (Two context, one switch, 150 000 input events)

Figure 5.6a and Figure 5.6b display the measurements of run time and events per second for the component window deriving. The run time and the events per second metrics are not changing drastically with a decrease of the window count size for windows created during the second context.



(a) Run time (Source: Own figure) (b) Events per second (Source: Own figure)

Figure 5.6: Window count size - Context deriving measurements (Two context, one switch, 150 000 input events)

5.3 Qualitative Analysis

This section provides a qualitative analysis of state-of-the-art models and implementation in the field of data stream processing. Those models were explained in Chapter 2. The Context-aware, Fact And Background Integrated Dynamic Stream Processing (CoFaBidSP) model presented in Chapters 3 and 4 is included in the analysis. The analysis is based on five criteria on a four-level rating scale. The used rating scale is displayed in Table 5.6.

Rating	**Meaning**
-	no integration/support
☆	can be integrated
☆☆	explicitly modeled/mentioned
☆ ☆ ☆	existing implementation

Table 5.6: Rating scale (Source: Own representation)

The criteria focus on two main categories. First, whether the model allows for integration of Context information, Facts and Background data. Second, whether models supports dynamic windowing and User-defined windows. Dynamic windowing means that the window-deriving adapts dynamically to changes in the context or generated facts. User-defined windows allow the user to influence the window-deriving process externally by changing the used Window size during run time. Table 5.7 shows the comparison of five selected models and implementations.

Model	Integration of			Support for	
	Context	**Facts**	**Background data**	**Dynamic windowing**	**User-defined windows**
Data Flow Model	☆	☆	☆	☆☆☆	☆☆☆
Frames	-	☆☆	-	☆☆	☆☆
CAESAR	☆☆☆	☆☆	-	☆☆	☆☆
TEF-SPAQRL	☆	☆☆☆	-	☆☆	☆☆
CoFaBidSP	☆☆☆	☆☆☆	☆☆	☆☆☆	☆☆☆

Table 5.7: Qualitative Analysis (Source: Own representation)

6

Limitations and future work

In this chapter I discuss the limitations of the proposed conceptual model as well as its implementation. Next, I propose concrete tasks to improve the current implementation as well as ideas on how to extend the conceptual model.

The proposed model defines the context, facts and background information as external components to stream processing system. This leads to additional integration efforts while programming data streaming pipelines. By increasing the number of contexts and context switches (more than in the conducted experiments) the overall performance could be affected due to frequent updates to the external component of the context. The same is valid for facts. Since the system design and implementation uses for the context, facts and background information centralized database tables, the run time performance could affect depending on the number of fact updates.

Depending on the definition of facts and the used window size, a high number of fact versions with very short validity periods are generated. For integrating events, facts and background information in the Analytics component the selection of "relevant" facts is not trivial.

The components Context deriving and Window Deriving are executed in parallel and no synchronization is intended. That means in the time period needed for updating the context information, the window processing component is using outdated context information for a short period of time (at maximum a few seconds). For most use cases it can be assumed that this delay could be acceptable.

The model is currently restricted to only one active context. This simplifies the implementation. Nevertheless multiple active contexts are existing in many uses cases, e.g. in the traffic management case study with "rush hour" and "day".

By using Apache Flink, the execution of the components can be executed on a cluster with multiple computers acting as different Task Managers. But the depending on the use case, the data used for the Window deriving component can not be partitioned, because it is necessary to use a global window for realizing data driven, user-defined windows.

For future work I propose the following concrete tasks. First, the distribution of context,

facts and background information in a distributed data base. The challenge is to synchronize the context, facts and background information inside a distributed data base with the distributed processing of the input events by Apache Flink.

Since Apache Flink 1.4 new functionality regarding state management is available (see Carbone et al., 2017). Further investigation on how to implement synchronization mechanisms between the distributed context, facts, background information and the distributed stream processing is recommended.

Second, designing and executing experiments in a fully distributed implementation of the conceptual model is useful to get a better understanding about the impacts of distributing the different components.

Third, benchmark this prototype with other implementations using the soccer monitoring case study. This would be helpful to compare the functionality and performance of the implemented prototype with other existing implementations. Furthermore it could lead to discover advantages and disadvantages of the design decisions and their implementations.

For the Window processing and Analytics components rough ideas are explained in Chapter 3 and Chapter 4. A formal foundation for integrating events with facts is provided in TEF-SPARQL (Kietz et al., 2013), but needs to enhanced for using in the proposed model and for implementing a solution by using the Table and SQL API of Apache Flink.

7

Conclusions

The generation of data is increasing rapidly. It is no longer produced only by companies or private people, but also by an increasing number of machines equipped with sensors. One key point is that the data is produced continuously, therefore processing the data should also be in a continuous way. This can be done by data stream processing systems, the first systems e.g. STREAM were developed in 2003. Today's use cases such as traffic management or soccer monitoring require models and systems which can adapt their processing based on changes in the application context and need to be able to integrate various information types such as context, facts and background information to deliver valuable near-real time insights.

Therefore this thesis focuses on two research questions related to this two main aspects of modern data stream processing:

First, how could a conceptual model for stream processing look like, which allows the combination of context-awareness, background knowledge and time-restricted facts? Chapter 3 presents a model to investigate event data, context, facts and background information in a comprehensive model. Furthermore, the processing model provides details about the components and their interaction with each other. It was shown that the model allows to detect a changed application context and adapt the data stream processing to the changed context.

Second, Chapter 4 described a system design and explained implementation details how such a conceptual model could be implemented, as asked in the second research question: What would be the architecture of an engine implementing such a model? The system is implemented with the Apache Flink Framework as stream processing system and a PostgreSQL database for the context and background information. A PostgreSQL extension for temporal databases is used to support time-restricted facts.

The model and implementation have been tested in a quantitative and qualitative analysis, described in Chapter 5. By using two metrics (run time and events per second), the influence of four parameters to the performance of the system was investigated. It has been shown that the integration of context-awareness, facts, background information does not decrease the overall performance. Nevertheless, the functionality is increased compared to other available implementations.

Chapter 6 provides an overview of concrete tasks as well as ideas for improving the presented conceptual model and implementation for context-aware stream processing.

Bibliography

Akidau, T. (2015). The world beyond batch: Streaming 101. Retrieved from https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., ... Whittle, S. (2015). The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow. 8*(12), 1792–1803. doi:10.14778/2824032.2824076

Ali, M. I., Gao, F., & Mileo, A. (2015). Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, ... S. Staab (Eds.), *The semantic web - iswc 2015* (pp. 374–389). Cham: Springer International Publishing.

Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A. S., Ryvkina, E., ... Tibbetts, R. (2004). Linear road: A stream data management benchmark. In *Proceedings of the thirtieth international conference on very large data bases - volume 30* (pp. 480–491). VLDB '04. Toronto, Canada: VLDB Endowment. Retrieved from http://dl.acm.org/citation.cfm?id=1316689.1316732

Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002). Models and issues in data stream systems. In *Proceedings of the twenty-first acm sigmod-sigact-sigart symposium on principles of database systems* (pp. 1–16). PODS '02. Madison, Wisconsin: ACM. doi:10.1145/543613.543615

Brőring, A., Remke, A., Stasch, C., Autermann, C., Rieke, M., & Mőllers, J. (2015). Envirocar: A citizen science platform for analyzing and mapping crowd-sourced car sensor data. *Transactions in GIS*, *19*(3), 362–376. doi:10.1111/tgis.12155

Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., & Tzoumas, K. (2017). State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow. 10*(12), 1718–1729. doi:10.14778/3137765.3137777

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, *36*(4).

Carbone, P., Traub, J., Katsifodimos, A., Haridi, S., & Markl, V. (2016). Cutty: Aggregate sharing for user-defined windows. In *Proceedings of the 25th acm international*

on conference on information and knowledge management (pp. 1201–1210). CIKM '16. Indianapolis, Indiana, USA: ACM. doi:10.1145/2983323.2983807

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., & Zdonik, S. B. (2003). Scalable distributed stream processing. In *Cidr* (Vol. 3, pp. 257–268).

Cugola, G. & Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, *44*(3), 15.

Dean, J. & Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, *51*(1), 107–113. doi:10.1145/1327452.1327492

DuCharme, B. (2013). *Learning sparql: Querying and updating with sparql 1.1*. Sebastopol: O'Reilly Media, Inc.

enviroCar. (2015). Envirocar. Retrieved January 12, 2018, from https://envirocar.org/

Grossniklaus, M., Maier, D., Miller, J., Moorthy, S., & Tufte, K. (2016). Frames: Data-driven windows. In *Proceedings of the 10th acm international conference on distributed and event-based systems* (pp. 13–24). ACM.

Helmer, S., Poulovassilis, A., & Xhafa, F. (2011). *Reasoning in event-based distributed systems*. Studies in computational intelligence. Berlin: Springer.

IBM. (2013). The four v's of big data. Retrieved February 3, 2018, from http://www.ibmbigdatahub.com/infographic/four-vs-big-data

Kaufmann, M., Fischer, P. M., May, N., & Kossmann, D. (2014). Benchmarking bitemporal database systems: Ready for the future or stuck in the past? In *Edbt* (Vol. 738, p. 749).

Kietz, J., Scharrenbach, T., Fischer, L., Bernstein, A., & Nguyen, K. (2013). *Tef-sparql: The ddis query-language for time annotated event and fact triple-streams*. Tech. Rep., Technical Report, University of Zurich, Department of Informatics.

Margara, A., Dell'Aglio, D., & Bernstein, A. (2017). Break the windows: Explicit state management for stream processing systems. In *Edbt* (pp. 482–485).

Marz, N. & Warren, J. (2015). *Big data: Principles and best practices of scalable realtime data systems*. Manning Publications Co.

Mutschler, C., Ziekow, H., & Jerzak, Z. (2013). The debs 2013 grand challenge. In *Proceedings of the 7th acm international conference on distributed event-based systems* (pp. 289–294). DEBS '13. Arlington, Texas, USA: ACM. doi:10.1145/2488222.2488283

Poppe, O., Lei, C., Rundensteiner, E. A., & Dougherty, D. J. (2016). Context-aware event stream analytics. In *Edbt* (pp. 413–424).

Snodgrass, R. T. (2000). *Developing time-oriented database applications in sql*. Morgan Kaufmann Publishers,

Tanenbaum, A. S. & Van Steen, M. (2007). *Distributed systems: Principles and paradigms* (2nd). Prentice-Hall.

A

Evaluation results

This section provides details about the conducted experiment runs. For every parameter configuration the program was executed three times. For the quantitative analysis only the average values were used. And only one parameter was changed, while the other parameters were hold constant.

ParameterConfiguration	File		Events	Start	End	StartTime	EndTime	RuntimeSeconds	Events/Second
NumberOfEvents									
1	full-game	context_default	150'000	22.03.2018 11:27:35	22.03.2018 11:28:02	1'521'714'455'801	1'521'714'482'550	26.00	5'769.00
1	full-game	context_default	150'000	22.03.2018 11:31:54	22.03.2018 11:32:23	1'521'714'714'616	1'521'714'743'711	29.00	5'172.00
1	full-game	context_default	150'000	22.03.2018 11:36:11	22.03.2018 11:36:41	1'521'714'971'301	1'521'715'001'311	30.00	5'000.00
								28.33	**5'313.67**
2	full-game	context_default	500'000	21.03.2018 15:23:19	21.03.2018 15:25:13	1'521'642'199'786	1'521'642'313'639	113.00	4'424.00
2	full-game	context_default	500'000	21.03.2018 20:50:18	21.03.2018 20:51:46	1'521'661'818'037	1'521'661'906'131	88.00	5'681.00
2	full-game	context_default	500'000	21.03.2018 21:01:34	21.03.2018 21:03:04	1'521'662'494'526	1'521'662'584'005	89.00	5'617.00
								96.67	**5'240.67**
3	full-game	context_default	1'000'000	22.03.2018 10:10:08	22.03.2018 10:13:24	1'521'709'808'060	1'521'710'004'970	196.00	5'102.00
3	full-game	context_default	1'000'000	22.03.2018 10:28:47	22.03.2018 10:31:48	1'521'710'927'469	1'521'711'108'527	181.00	5'524.00
3	full-game	context_default	1'000'000	22.03.2018 10:53:42	22.03.2018 10:56:29	1'521'712'422'411	1'521'712'589'581	167.00	5'988.00
								181.33	**5'538.00**
4	full-game	context_default	2'000'000	23.03.2018 15:22:42	23.03.2018 15:29:18	1'521'814'962'576	1'521'815'358'075	395.00	5'063.00
4	full-game	context_default	2'000'000	22.03.2018 14:36:01	22.03.2018 14:42:21	1'521'725'761'593	1'521'726'141'492	379.00	5'277.00
4	full-game	context_default	2'000'000	23.03.2018 15:57:22	23.03.2018 16:03:51	1'521'817'042'649	1'521'817'431'821	389.00	5'141.00
								387.67	**5'160.33**
13	full-game	context_default	3'000'000	23.03.2018 12:52:57	23.03.2018 13:01:34	1'521'805'977'767	1'521'806'494'235	516.00	5'813.00
13	full-game	context_default	3'000'000	23.03.2018 13:03:59	23.03.2018 13:12:41	1'521'806'639'167	1'521'807'161'253	522.00	5'747.00
13	full-game	context_default	3'000'000	23.03.2018 13:13:44	23.03.2018 13:24:15	1'521'807'224'460	1'521'807'855'278	630.00	4'761.00
								556.00	**5'440.33**
14	full-game	context_default	4'000'000	23.03.2018 13:26:08	23.03.2018 13:40:08	1'521'807'968'071	1'521'808'808'929	840.00	4'761.00
14	full-game	context_default	4'000'000	23.03.2018 13:42:57	23.03.2018 13:53:55	1'521'808'977'982	1'521'809'635'820	657.00	6'088.00
14	full-game	context_default	4'000'000	23.03.2018 14:12:05	23.03.2018 14:25:24	1'521'810'725'560	1'521'811'524'651	799.00	5'006.00
								765.33	**5'285.00**
5	full-game	context_default	5'000'000	21.03.2018 15:52:51	21.03.2018 16:12:29	1'521'643'971'007	1'521'645'149'127	1'178.00	4'244.00
5	full-game	context_default	5'000'000	21.03.2018 22:27:25	21.03.2018 22:42:35	1'521'667'645'922	1'521'668'555'225	909.00	5'500.00
5	full-game	context_default	5'000'000	23.03.2018 15:03:42	23.03.2018 15:19:55	1'521'813'822'162	1'521'814'795'484	973.00	5'138.00
								1'020.00	**4'960.67**

Figure A.1: NumberOfEvents Context Experiments (Source: Own representation)

ParameterConfiguration	Datei	Context	Events	Start	End	StartTime	EndTime	RuntimeSeconds	Events/Second
Number of Events									
1	full-game	1	150'000	22.03.2018 11:27:41	22.03.2018 11:28:10	1'521'714'461'210	1'521'714'490'055	28.00	5'357.00
1	full-game	1	150'000	22.03.2018 11:31:58	22.03.2018 11:32:27	1'521'714'718'000	1'521'714'747'808	29.00	5'172.00
1	full-game	1	150'000	22.03.2018 11:36:15	22.03.2018 11:36:45	1'521'714'975'172	1'521'715'005'118	29.00	5'172.00
								28.67	**5'233.67**
2	full-game	1	500'000	21.03.2018 20:50:23	21.03.2018 20:51:41	1'521'661'823'144	1'521'661'901'620	78.00	6'410.00
2	full-game	1	500'000	21.03.2018 20:57:02	21.03.2018 20:58:21	1'521'662'222'974	1'521'662'301'057	78.00	6'410.00
2	full-game	1	500'000	21.03.2018 21:01:47	21.03.2018 21:03:13	1'521'662'507'148	1'521'662'593'308	86.00	5'813.00
								80.67	**6'211.00**
3	full-game	1	1'000'000	22.03.2018 10:10:11	22.03.2018 10:12:58	1'521'709'811'975	1'521'709'978'803	166.00	6'024.00
3	full-game	1	1'000'000	22.03.2018 10:28:51	22.03.2018 10:31:15	1'521'710'931'921	1'521'711'075'296	143.00	6'993.00
3	full-game	1	1'000'000	22.03.2018 10:53:47	22.03.2018 10:56:22	1'521'712'427'156	1'521'712'582'597	155.00	6'451.00
								154.67	**6'489.33**
4	full-game	1	2'000'000	23.03.2018 15:22:48	23.03.2018 15:29:01	1'521'814'968'992	1'521'815'341'418	372.00	5'376.00
4	full-game	1	2'000'000	2018-03-22 14:46:53.024	22.03.2018 14:51:45	1'521'726'413'024	1'521'726'705'364	292.00	6'849.00
4	full-game	1	2'000'000	23.03.2018 15:57:31	23.03.2018 16:03:41	1'521'817'051'882	1'521'817'421'559	369.00	5'420.00
								344.33	**5'881.67**
13	full-game	1	3'000'000	23.03.2018 12:53:06	23.03.2018 13:00:44	1'521'805'986'197	1'521'806'444'062	457.00	6'564.00
13	full-game	1	3'000'000	23.03.2018 13:04:03	23.03.2018 13:12:01	1'521'806'643'377	1'521'807'121'505	478.00	6'276.00
13	full-game	1	3'000'000	23.03.2018 13:13:52	23.03.2018 13:23:37	1'521'807'232'698	1'521'807'817'788	585.00	5'128.00
								506.67	**5'989.33**
14	full-game	1	4'000'000	23.03.2018 13:26:23	23.03.2018 13:39:42	1'521'807'983'464	1'521'808'782'812	799.00	5'006.00
14	full-game	1	4'000'000	23.03.2018 13:59:42	23.03.2018 14:10:20	1'521'809'982'975	1'521'810'620'273	637.00	6'279.00
14	full-game	1	4'000'000	23.03.2018 14:12:10	23.03.2018 14:24:16	1'521'810'730'282	1'521'811'456'157	725.00	5'517.00
								720.33	**5'600.67**
5	full-game	1	5'000'000	21.03.2018 21:22:45	21.03.2018 21:36:17	1'521'663'765'466	1'521'664'577'199	811.00	6'165.00
5	full-game	1	5'000'000	21.03.2018 21:44:23	21.03.2018 21:58:03	1'521'665'063'789	1'521'665'883'079	819.00	6'105.00
5	full-game	1	5'000'000	23.03.2018 15:03:49	23.03.2018 15:19:09	1'521'813'829'828	1'521'814'749'949	920.00	5'434.00
								850.00	**5'901.33**

Figure A.2: NumberOfEvents Window Experiments (Source: Own representation)

47

Number of contexts and contexts switches									
Two Context, One Switch									
7	full-game	context2	150'000	22.03.2018 11:38:01	22.03.2018 11:38:30	1'521'715'081'767	1'521'715'110'790	29.00	5'172.00
7	full-game	context2	150'000	22.03.2018 14:01:50	22.03.2018 14:02:20	1'521'723'710'383	1'521'723'740'068	29.00	5'172.00
7	full-game	context2	150'000	22.03.2018 11:42:47	22.03.2018 11:43:16	1'521'715'367'697	1'521'715'396'633	28.00	5'357.00
								28.67	**5'233.67**
8	full-game	context2	500'000	21.03.2018 20:29:03	21.03.2018 20:31:03	1'521'660'543'423	1'521'660'663'792	120.00	4'166.00
8	full-game	context2	500'000	21.03.2018 20:40:45	21.03.2018 20:42:09	1'521'661'245'083	1'521'661'329'939	84.00	5'952.00
8	full-game	context2	500'000	21.03.2018 20:44:38	21.03.2018 20:46:14	1'521'661'478'358	1'521'661'574'689	96.00	5'208.00
Three Context, Two Switch									
9	full-game	context3	150'000	22.03.2018 11:46:27	22.03.2018 11:46:57	1'521'715'587'813	1'521'715'617'273	29.00	5'172.00
9	full-game	context3	150'000	22.03.2018 13:06:46	22.03.2018 13:07:18	1'521'720'406'117	1'521'720'438'916	32.00	4'687.00
9	full-game	context3	150000	22.03.2018 13:16:32	22.03.2018 13:17:00	1'521'720'992'211	1'521'721'020'941	28.00	5'357.00
Four Context, Three Switch								**29.67**	5'072.00
10	full-game	context4	150'000	22.03.2018 14:12:39	22.03.2018 14:13:09	1'521'724'359'548	1'521'724'389'163	29.00	5'172.00
10	full-game	context4	150'000	22.03.2018 13:42:52	22.03.2018 13:43:22	1'521'722'572'374	1'521'722'602'507	30.00	5'000.00
10	full-game	context4	150'000	22.03.2018 13:42:52	22.03.2018 13:43:22	1'521'722'572'374	1'521'722'602'507	30.00	5'000.00
								29.67	**5'057.33**

Figure A.3: ContextSwitches Context Experiments (Source: Own representation)

Number of contexts and contexts switches									
Two Context, One Switch									
7	full-game	2	150'000	22.03.2018 11:38:08	22.03.2018 11:38:36	1'521'715'088'350	1'521'715'116'680	28.00	5'357.00
7	full-game	2	150'000	22.03.2018 11:41:28	22.03.2018 11:41:56	1'521'715'288'353	1'521'715'316'160	27.00	5'555.00
7	full-game	2	150'000	22.03.2018 11:42:51	22.03.2018 11:43:20	1'521'715'371'203	1'521'715'400'230	29.00	5'172.00
								28.00	**5'361.33**
8	full-game	2	500'000	21.03.2018 20:29:11	21.03.2018 20:31:02	1'521'660'551'888	1'521'660'662'637	110.00	4'545.00
8	full-game	2	500'000	21.03.2018 20:40:54	21.03.2018 20:42:18	1'521'661'254'049	1'521'661'338'757	84.00	5'952.00
8	full-game	2	500'000	21.03.2018 20:44:42	21.03.2018 20:46:05	1'521'661'482'469	1'521'661'565'066	82.00	6'097.00
Three Context, Two Switch									
9	full-game	3	150'000	22.03.2018 11:46:35	22.03.2018 11:47:04	1'521'715'595'459	1'521'715'624'101	28.00	5'357.00
9	full-game	3	150'000	22.03.2018 13:06:49	22.03.2018 13:07:20	1'521'720'409'113	1'521'720'440'165	31.00	4'838.00
9	full-game	3	150'000	22.03.2018 13:16:36	22.03.2018 13:17:06	1'521'720'996'283	1'521'721'026'449	30.00	5'000.00
								29.67	**5'065.00**
10	full-game	4	150'000	22.03.2018 13:39:46	22.03.2018 13:40:16	1'521'722'386'411	1'521'722'416'266	29.00	5'172.00
10	full-game	4	150'000	22.03.2018 13:43:00	22.03.2018 13:43:27	1'521'722'580'648	1'521'722'607'827	27.00	5'555.00
10	full-game	4	150'000	22.03.2018 14:12:44	22.03.2018 14:13:12	1'521'724'364'097	1'521'724'392'407	28.00	5'357.00
								28.00	**5'361.33**

Figure A.4: ContextSwitches Window Experiments (Source: Own representation)

49

Two Context, One Switch, DifferentWindowCountSize										
11	full-game	context5	150'000	22.03.2018 15:25:44	22.03.2018 15:26:13	1'521'728'744'664	1'521'728'773'813	29.00	5'172.00
11	full-game	context5	150'000	22.03.2018 15:47:20	22.03.2018 15:47:48	1'521'730'040'823	1'521'730'068'724	27.00	5'555.00
11	full-game	context5	150'000	22.03.2018 15:51:42	22.03.2018 15:52:12	1'521'730'302'640	1'521'730'332'230	29.00	5'172.00
								28.33	**5'299.67**
12	full-game	context6	150'000	22.03.2018 15:32:38	22.03.2018 15:33:05	1'521'729'158'065	1'521'729'185'313	27.00	5'555.00
12	full-game	context6	150'000	22.03.2018 15:36:34	22.03.2018 15:37:01	1'521'729'394'540	1'521'729'421'840	27.00	5'555.00
12	full-game	context6	150'000	22.03.2018 15:38:58	22.03.2018 15:39:27	1'521'729'538'071	1'521'729'567'247	29.00	5'172.00
								27.67	**5'427.33**

Figure A.5: WindowCountSize Context Experiments (Source: Own representation)

Two Context, One Switch, DifferentWindowCountSize									
11	full-game	5	150'000	22.03.2018 15:19:07	22.03.2018 15:19:34	1'521'728'347'028	1'521'728'374'380	27.00	5'555.00
11	full-game	5	150'000	22.03.2018 15:23:35	22.03.2018 15:24:02	1'521'728'615'654	1'521'728'642'691	27.00	5'555.00
11	full-game	5	150'000	22.03.2018 15:47:30	22.03.2018 15:47:59	1'521'730'050'729	1'521'730'079'295	28.00	5'357.00
								27.33	**5'489.00**
11	full-game	6	150'000	22.03.2018 15:32:45	22.03.2018 15:33:13	1'521'729'165'435	1'521'729'193'267	27.00	5'555.00
11	full-game	6	150'000	22.03.2018 15:36:44	22.03.2018 15:37:12	1'521'729'404'750	1'521'729'432'572	27.00	5'555.00
11	full-game	6	150'000	22.03.2018 15:39:02	22.03.2018 15:39:32	1'521'729'542'953	1'521'729'572'461	29.00	5'172.00
								27.67	**5'427.33**

Figure A.6: WindowCountSize Window Experiments (Source: Own representation)

51

B

Contents of the CD

This section lists the content of the CD attached to this thesis.

- **Abstract.txt**
  English version of the abstract
- **flink-1.4.1**
  Standalone Apache Flink cluster (including the data file, named: full-game)
- **flink-quickstart-java**
  Java source files used for this master thesis
- **Latex**
  Latex source files of this master thesis
- **ReadMe.pdf**
  Technical installation guidelines to install and run the implemented prototype
- **soccer_monitoring.db**
  PostgreSQL dump of database soccer_monitoring
- **Masterarbeit.pdf**
  PDF version of this master thesis
- **Zusfsg.txt**
  German version of the abstract

C

Installation Guidelines

This section states a checklist for installing and running the implemented prototype of the Context-aware, Fact And Background Integrated Dynamic Stream Processing (CoFaBidSP) model. Further information can be also found in the ReadMe.pdf on the attached CD.

1. Optional, but recommended: Usage of Ubuntu (or other Linux-based OS)

2. Install Java 1.8

3. Install Apache Flink 1.4.1 as Standalone Cluster
   Using the files provided on the CD or via https://flink.apache.org/

4. Recommended: Install IntelliJ IDEA Community Edition 2017.2
   https://www.jetbrains.com/idea/download/

5. Install PostgreSQL 9.5.10 (User: postgres, Password: postgres)
   `sudo apt-get install postgresql-9.5`

6. Install PostgreSQL temporal table extension 1.2.0
   `sudo pgxn install_temporal tables`
   For more details see ReadMe.pdf on the CD attached to this thesis.

7. Create database soccer_monitoring in PostgreSQL
   `psql -h localhost -U postgres CREATE DATABASE soccer_monitoring`

8. Import PostgreSQL dump file
   `psql -h localhost -U postgres soccer_monitoring < soccer_monitoring.db`

9. Import the source files as a Java Project in IntelliJ IDEA
   Click on “File → New → Project from existing source”.
   Select the folder “flink-quickstart-java” of the attached CD

10. Optional: Build a JAR File of the project

11. Run JAR-File in the folder “flink-1.4.1” on the local Flink Cluster via Terminal
    Open two separate terminals, start the context deriving first, and shortly after that the window deriving.

List of Figures

List of Tables