Grand Challenge: The TechniBall System

Avigdor Gal, Sarah Keren, Mor Sondak, Matthias Weidlich
Technion - Israel Institute of Technology
avigal@ie.technion.ac.il
{sarahn,mor,weidlich}@tx.technion.ac.il

Hendrik Blom, Christian Bockermann
TU Dortmund
hendrik.blom@tu-dortmund.de,
christian.bockermann@udo.edu

ABSTRACT

In this work we present the solution to the DEBS'2013 Grand Challenge, as crafted by the joint effort of teams from the Technion and TU Dortmund. The paper describes the architecture, details the queries, shows throughput and latency evaluation, and offers our observations regarding the appropriate way to trade-off high-level processing with time constraints.

Categories and Subject Descriptors

H.3.4 [**System and Software**]: Distributed Systems

Keywords

event processing, real-time event processing

1. INTRODUCTION

The ACM DEBS Grand Challenge series seeks to provide a common ground and evaluation criteria for event-based solutions, by offering problems that require event-based systems and that can be evaluated using real-life data and queries. The 2013 challenge [16] involves real-time complex analytics over high velocity sensor data using the example of analyzing a soccer game. The input data comes from sensors in player shoes and a ball used during a soccer match. The real-time analytics involves continuous computation of four main statistics types, namely ball possession, shots on goal, heat maps, and running analysis of team members.

This paper presents the *TechniBall* solution that was developed at the Technion – Israel Institute of Technology together with the Technical University in Dortmund. The proposed solution strikes a balance between the use of high-level complex event processing language (Esper [9]) and a fast low-level processing of events that arrive at the pico-second level (*streams* [5]).

The paper is structured as follows: In Section 2 we detail the overall architecture and discuss implementation aspects. Section 3 presents the queries that support the required analytics. We share our experiences in building and running the system in Section 4 and conclude with related work discussion (Section 5) and future work (Section 6).

DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.

2. ARCHITECTURE

The TechniBall solution is based on a shell for defining data flows and manages parallelization. From within the shell, events undergo initial processing and are then routed to a CEP engine for pattern recognition (see Figure 1).

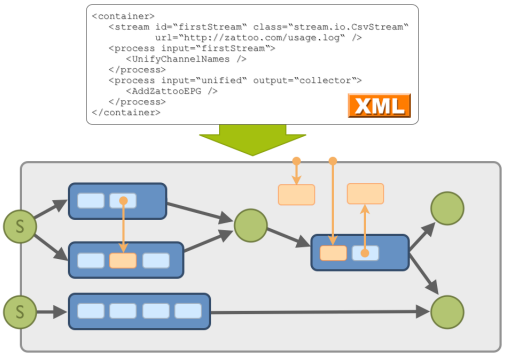

Figure 1: Generic architecture description

For the shell, we used the open-source *streams* framework [6], which provides a description language consisting of sources, sinks and processors. The data flow is defined in high-level XML and is compiled into a computation graph for a stream processing engine (e.g., the *streams runtime* or a topology for the *Storm* engine [15]). *Streams* provides a clean and easy-to-use Java-based middleware to design and implement data stream processes on different stream processing engines. It promotes simple software design patterns such as JavaBean conventions and dependency injection to allow for a quick setup of streaming processes. *Streams* also supports rapid prototyping and allows for direct integration of custom processor classes into the configuration. TechniBall uses the latest *streams* version 0.9.10-SNAPSHOT.

As the CEP engine, we were looking for a platform that easily integrates into a java project and supports the needed expressive power. We chose Esper [9] following the recommendations in [7]. Esper enables rapid development of applications that process large volumes of incoming events in real-time. Esper filters and analyzes events using a declarative high-level, SQL-like language, responding to conditions of interest with minimal latency. We used Esper 4.6.0 for

Java, open-source software available under the GNU General Public License (GPL). We implemented a single custom Esper processor using the *streams* API, directly including Esper queries into the XML description of the data flow graphs using a simple `esper.Query` tag. Figure 2 shows the definition of a computation graph with a single input stream and a process that includes the Esper process.

```
<container id="sample-config">
  <properties>
    <!-- define attribute types of attributes
         used in ESPER queries below -->
    <esper.types>...</esper.types>
  </properties>

  <!-- define a stream implemented by the custom soccer stream
       class that has been optimized for the DEBS data  -->
  <stream id="soccer" url="file:/path/to/game-data"
       class="stream.io.FastSoccerStream" />

  <!--  define a process node connected to the soccer stream
        that applies the esper engine and send results to the
        queue "R"                          -->
  <process input="soccer">
    <stream.esper.Query output="R" types="${esper.types}">
            Esper QUERY
    </stream.esper.Query>
  </process>

  <!--  define a process connected to "R" that simply prints
        out all items to stdout    -->
  <process input="R">
     <PrintData />
  </process>
</container>
```

Figure 2: Esper query embedded in *streams*

The TechniBall implementation comes as a single executable Java archive that includes the *streams* runtime and the Esper engine. It receives as input a data flow graph definition in an XML file. For the DEBS challenge we crafted distributed processing of multiple threads. To comply with the challenge specification, we created a virtual machine image running Ubuntu Server Edition 12.10 (64bit, kernel 3.5.0). We tested this image with a virtual machine that features the predefined 4 cores (@2.8 GHz) and 4 GB RAM. Figure 3 (next page) provides an overall architecture of the TechniBall system. The *streams* framework provides stream and queue implementations (circles) and the notion of processes (dark shade boxes), which read from these and execute a list of processors (light shade boxes) for each item read. The top left stream, `S`, represents the data input. Items are read from the stream as simple hash-maps, which can contain arbitrary serializable values, allowing for dynamically adding new fields to the data that can be consumed by subsequent processors.
A single reader process reads in the data and applies a set of operations (processors) to each element of the stream. These elements provide pre-processing such as joining meta-information to each sensor measurement (player ID, sensor type). The final processor of the reader process dispatches copies of the sensor measurements to several queues, each of which is processed by its own process. The modularity of the solution allows processing several queries in a single thread. The shaded optional processor `GameView` can be added to display a live view of the game (see Figure 12).

3. CHALLENGE QUERIES

We now turn our attention to the four queries posed by the challenge. For most of the query implementations, the input data events need to be preprocessed by applying a chain of processors that have been implemented using the *streams* API. The data pre-processing enriches the data using a metadata file, which contains the player names, associates the transmitter IDs with a player ID (`pid`) and adds fields for the type of sensor (leg, arm, left, right). The player ID provided by the metadata is artificially set up and numbers the players from 0 to 17, where ID 0 is assigned to the referee sensors and all ball sensors are assigned a negative ID. In addition, all players of team A are assigned odd IDs and the players of team B are equipped with even IDs. This allows (a) for using arrays as data structures with the `pid` as the index to player data and (b) for using the *modulo* operator to determine to which team the player belongs. Other operators add flags for signaling whether the game is currently active or compute the player from the leg sensors. The enriched events are marked as $E1$ in Figure 3.
After pre-processing, the enriched events are distributed into three queues, Q_1, Q_2, and Q_3 that are processed by separate threads, each of which handles a different query. The shot detection is part of query 2 and is required in query 4, so queue Q_4 is fed with the detected shots. Query results are assigned a reference and sent to result queue R, from which a process reads and emits the items to standard output.

3.1 Query 1 – Running Analysis

Query 1 keeps track of each players current running distance and reports on the intensity of the runs (e.g. trotting). The process for the running analysis is shown in Figure 4.

```
<process input="Q:1">
    <!--  Track current intensity for each player -->
    <stream.soccer.AddIntensity />

    <!-- Aggregate the intensity levels for each player
         and emit results to the queue "R"      -->
    <stream.soccer.AggregateCurrentIntensity output="R" />
</process>
```

Figure 4: The process definition for the running analysis.

We implemented a processor that keeps track of the intensity level for each player. According to the challenge requirements, intensities need to stay stable over a full second before they are regarded as a new intensity state. Our `AddIntensity` processor implements this tracking of player intensities, aggregates the information of all sensors for each player, and emits the stable intensity levels. For computing the aggregated intensities, we implemented two alternative approaches: First, we created an Esper query to aggregate the events directly. The query in Figure 5 performs aggregation for the trot level for a 1min time window:

```
<esper.Query condition="%{data.pid} @gt 0"  output="R">
  SELECT pid AS player,
    sum(case when intensity=1 then distance else 0 end)
          as trot_distance,
    sum(case when intensity=1 then (ts_stop - ts_start)
          /1000000000 else 0 end) as trot_time
    FROM Data.win:time(60000 sec) GROUP BY pid
</esper.Query>
```

Figure 5: Esper implementation of Query 1

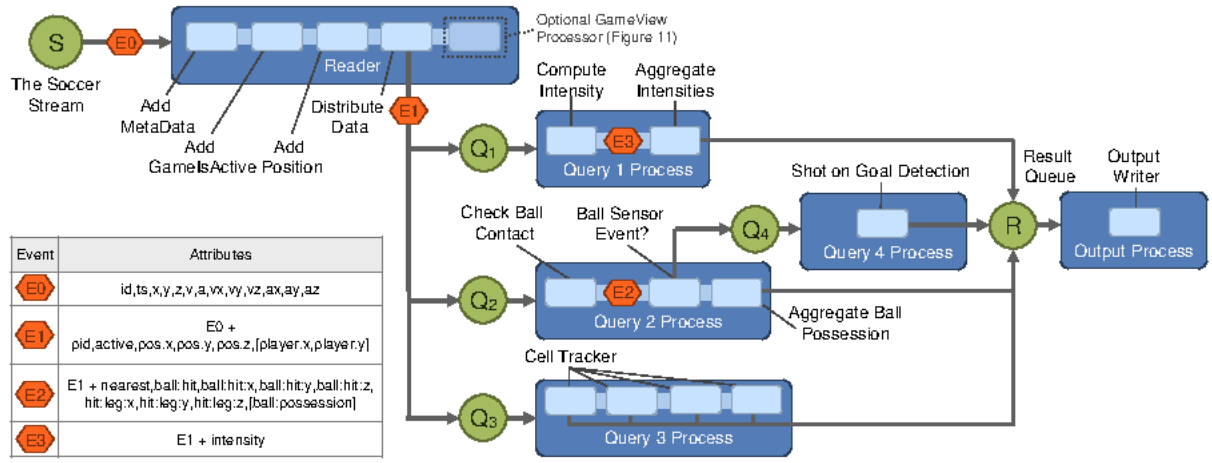

Figure 3: TechniBall overall architecture

An alternative solution directly implemented the aggregation as a separate processor `AggregateCurrentIntensity` using the *streams* API, exploiting some of the features added in the pre-processing step such as direct array lookup of data structures using the `pid` value.

3.2 Query 2 – Ball Possession

The ball possession query requires information from several event types: The player current positions need to be handled and maintained; the ball position needs to be assessed; and the condition for a ball hit needs to be constantly checked. Finally, the ball possession status needs to be maintained and the times of possession need to be aggregated according to the windows sizes. Figure 6 shows the XML definition of the process that handles the ball possession query.

```
<process input="Q:2">
    <!--  Add details on hits of the ball -->
    <stream.soccer.CheckBallContact />

    <!-- Handle possession changes and aggregate the
         possession times for the required windows    -->
    <stream.soccer.AggregateBallPossession output="R" />

    <!--  Send ball events for shot on goal detection -->
    <Enqueue condition="%{data.pid} @lt 0" queue="Q:4" />
</process>
```

Figure 6: Ball possession query

The `CheckBallContact` processor keeps track of the current player positions and checks whether the *ball hit* condition is fulfilled for any incoming ball event. Keeping track of players is done in constant time using the `pid` value from the pre-processing step as index to an array holding player positions. The *ball hit* condition is only checked for ball sensors which are within the field boundaries, thus, this processor automatically chooses the active ball for its computations. If a ball hit is detected, the `ball:hit` field of the event is set to the `pid` of the player that hit the ball. In addition, this processor maintains the current possessor of the ball. Once the possessor changes, the possession time of the last player is emitted. If the ball leaves the field or the game is interrupted, possession is transferred to player with `pid` 0 (referee). The processor `AggregateBallPossession` maintains sliding windows for each of the requested windows and aggregates the possession times provided by the previous processor. Upon changes of the possession, the current aggregations are emitted to the result queue. An alternative version for this aggregation can be performed by the Esper query shown in Figure 7 (showing only the aggregation by player as an example).

```
<process input="Q:2">
  <stream.soccer.CheckBallContact />

  <!-- Use Esper to aggregate the ball hits grouped
       by player over a sliding window      -->
  <stream.esper.Query condition="%{data.ball:hit} > 0" output="R">
    SELECT max(ts) AS ts, pid AS player,
      sum(‘ball:possession‘) AS time,
      sum(‘ball:hit‘) AS hits
    FROM Data GROUP BY pid OUTPUT LAST EVERY 1 EVENTS;
   </stream.esper.Query>
   <Enqueue condition="%{data.pid} @lt 0" queue="Q:4" />
</process>
```

Figure 7: Esper aggregation of ball possession

3.3 Query 3 - Heat Map

The heat map tracking is directly implemented in a Java processor within *streams*. `CellTracker` allows for specifying the granularity of the grid and pre-initializes the memory required to track all cells for each player. Each of the cell objects allocates memory for keeping the aggregation windows at initialization time as well. This is a direct memory-to-speed tradeoff – by pre-allocating the memory and using a customized data structure, we ensure updates of our heatmaps in $\mathcal{O}(1)$: With the player ID (pid), added in the pre-processing phase, the cell updates for the grid for each event is then possible in constant time. From the x and y coordinates of the sensor we directly compute the index of the cell and the player ID is then used as the index of the counts that need to be updated. The output format of `CellTracker` is slightly different from the format proposed in the challenge description: instead of emitting an item for each cell (one

line per cell), we emit a compressed form of the cell array as

$$cell\_times : [\ldots, i : v_i, (i+1) : v_{i+1}, \ldots] \; \forall \, i : v_i > 0.$$

Thus, we emit the index of the cell and its value for all entries larger than 0. The rest of the provided fields of the emitted result items for the heat map queries match the format requested in the challenge description. Figure 8 provides one instance of `CellTracker` for each grid resolution.

```
<process input="Q:3">
   <soccer.CellTracker gridx="8" gridy="13" output="R" />
   <soccer.CellTracker gridx="16" gridy="25" output="R" />
   <soccer.CellTracker gridx="32" gridy="50" output="R" />
   <soccer.CellTracker gridx="64" gridy="100" output="R" />
</process>
```

Figure 8: Heat map with the four cell tracker instances

3.4 Query 4 – Shot on Goal

The shot on goal query requires a shot/hit of the ball as a trigger and then checks if the ball is directed towards the opponents goal. Processing this query is done together with the ball possession query, as this query already emits information about detected hits.
The `ShotOnGoal` processor computes the future position of a ball based on such a *ball hit* event and checks whether this position passes the goal areas as specified in the challenge description within the pre-defined amount of time. The future position is projected from the current position based on the direction and speed of the ball (including gravity). Essentially, the respective processor can be in three states, *initial*, *hit detected*, and *shot-on-goal detected*. We transition from *initial* to *hit detected* if the `ball:hit` flag is set for a ball event, recording the position of the leg that hit the ball. As soon as the ball is more than 1m away from this position, we check whether it would reach the goal accordingly and, if so, transition from *hit detected* to *shot-on-goal detected*. Whenever being in this state, we emit the relevant event and check whether the conditions for terminating the shot-on-goal are satisfied. If so, we transition to state *initial*.

3.5 Query Result Formats

To meet the challenge requirements regarding the result output formats, we provide a single output process, which prints out the result items emitted by the different query implementations to standard output. This leads to writing mixture of result items from different queries to a single standard output. Two additions allow simple inspection and parsing. First, we add a field `query` to each emitted line, which references the query for which this item has been emitted. Secondly, we encode each item as a JSON object and separate items by a line break. The JSON encoding allows adding fields. For example, for the heat maps, we emit items with the additional fields `query`, `grid` and `win`, which identify the query, the grid resolution and the window, to which the output line belongs.

4. EVALUATION

In this section we outline our evaluations of the *TechniBall* system. All experiments were performed using the OpenJDK Java virtual machine version 1.7.0_21 on an Intel Core i7-3770 CPU machine with 3.4 GHz with 16 GB of main memory. The game data is read from a ramdisk to eliminate disk latency. Java was started with 4 GB of heap space.

4.1 Reading and Parsing

To optimize the reader implementation to the data at hand, we compared different implementations for reading and parsing events from the file into objects. Java's line reader is capable of reading about 3.3 million lines per second – parsing this into numbers is the hard part. As can be seen in Figure 9(left), the default `split` and parse number functions results in rather poor performance. Using the StringTokenizer and custom parsing improves this. `FastSoccerStream` reads lines and directly parses numbers without splitting the lines into substrings. This leads to a data rate of about 922.000 events per second.

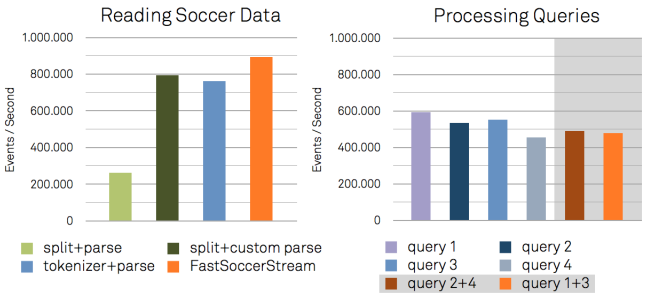

Figure 9: Reading and processing throughput

4.2 Processing Queries

With a basic knowledge of the upper bound of simply reading the data, we investigated the throughput of processing the challenge queries. Here, the combined use of *streams* and Esper was used with an attempt to gain the best of both worlds. We have built upon the abilities of *streams* to perform fast processing of enrichment, filtering, and computation. Esper, on the other hand, kicked in as soon as complex reasoning was needed, making use of its high level query language.
We have implemented queries using both Esper queries and *streams* code, trying out the trade-offs of declarative application specifications, rapid programming, maintainability, execution timing requirements, etc. The heavy emphasis on throughput in the challenge led the implementation away from Esper queries, in the direction of *streams* coding. In the remainder, we report on the performance obtained purely with the custom implementations in *streams*. Figure 9 shows the throughput for each of the queries: For each of those measurements, we set up a single process that reads from the `FastSoccerStream`, joins the meta-data (player ID,...) and applies some basic computations. After that we added the query processor and ran the setup. This was done for each query processor separately. The throughput (events per second) of running each query are shown in Figure 9 (right). Some of the queries work on the same events (e.g., running analysis and heat map do not require ball events) we also tested the throughput of combining queries 1 and 3 (running analysis and heat map) and queries 2 and 4 (ball possession and shot-on-goal). The combined throughput is shown in the shaded part of Figure 9 (right).

In addition we were interested on the latency induced by the query processing. As latency measurement, we computed the time it takes for an event to traverse the data flow graph until it is integrated into the statistics. The time was measured using the Java wall-clock time, which has a granularity of nanoseconds. Figure 10 shows the average latency induced by each of the queries. The average for latency for query processing is at 2356,512 nanoseconds per event.

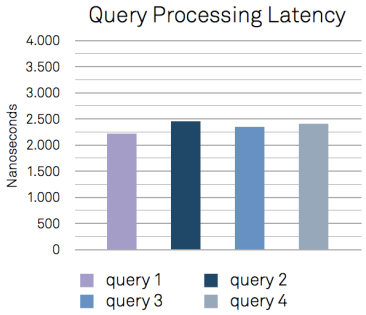

Figure 10: Latency induced by queries (in nanoseconds).

4.3 Going Parallel

Next we tested the throughput of parallel processing of all queries according to the setup of Figure 3 (*standard setup*). In addition we were interested in scaling up the throughput with different combinations of queries within the data flow setup. The *standard setup* features a single queue for each query. The other extreme is a single process that manages all queries (*single-threaded*). This setup is faster due to the required queue-synchronization in the *standard setup*, but does not provide the best overall throughput. By combining queries (e.g., q1+q3) within one process, we add parallelization with fewer synchronization overhead. In our experiments, the setup *q1+q2 || q2+q4* showed the best overall performance, with an average of about 281.000 events per second.

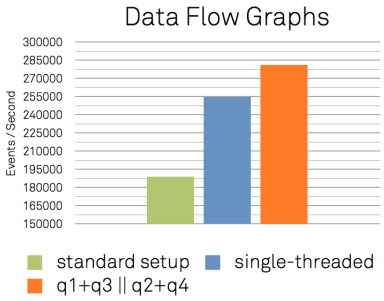

Figure 11: Parallelization impact on throughput

4.4 Data Visualization

Albeit the focus on pure throughput measurements, we have also been interested in a feedback of the raw data by means of a visual representation of the sensor measurements. For this, we implemented a simple `GameView` processor that displays the data with a soccer field visualization (see Figure 12). This feature highlights the player that possesses the ball and shot-detections, allowing the assessment of output statistics like ball possession while "watching the game."

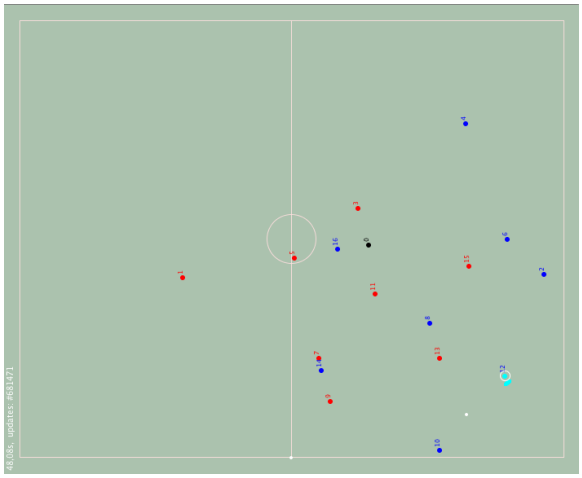

Figure 12: Visualization by the `GameView` processor

5. RELATED WORK

Our work combines a stream processing run-time with a CEP engine, thereby following the conceptual model of an event processing network as put forward by Etzion and Niblett [10]. Here, event processing agents are realized either as *streams* processors or Esper queries. Event channels are implemented by the structure of the *streams* data flow graph. As such, our work follows a stream-oriented model for event processing, rooted in Kahn process networks [12] for non-blocking data flow, so that event streams are modeled by event tuples.

Recently, various data stream engines have been proposed in the literature, such as Storm [15], S4.io [17], MOA [4] and Apache Kafka [14]. The *streams* framework [5, 6] has been designed as a middle-layer API that allows for an implementation of components for stream processing while being able to use these components as building blocks for designing data flow graphs that can be executed with the *streams-runtime* or compiled into Storm topologies. For example, the processors we implemented for the DEBS challenge, can directly be wrapped into Storm bolts (wrapping into bolts is provided by *streams*). As the DEBS challenge is focused on a single stream of data and a single processing node, we decided for the use of the *streams-runtime* as there is no need for the distribution features provided by Storm. *streams* can integrate MOA classifiers as building blocks that can be added to the stream graph. Likewise, we implemented processors for specifying Esper queries within the XML.

Turning to the definition of event processing agents, the Esper Query Language (EQL) is inspired by early query languages such as CQL [2] that declaratively describe the input streams, relevant event contexts, and event production. Our choice for Esper was motivated by a recent survey [7]. However, queries may also be defined based on active rules or logic programming. Active rules (aka Event-Condition-Action rules) are explicitly triggered by event occurrences. Triggers are defined in event algebra featuring operators such as 'sequence' and 'and' as done, e.g., in Snoop [8]. Query

languages based on logic programming employ temporal and action logics. The Event Calculus [13] and derived dialects [3] define axioms that relate to point-based events and the initialization and termination of fluents, properties that change with the occurrence of events. The Etalis system [1] is an example for an implementation of this model.

6. CONCLUSIONS

The paper presents a solution to the DEBS 2013 grand challenge. The solution is based on query parallelization and data stream flow control, using *streams* and Esper.

The proposed solution is based on the framework developed for the INSIGHT European project.[1] The goal of the INSIGHT project is to radically advance our ability of coping with emergency situations in smart cities by developing innovative technologies, methodologies, and systems that will put new capabilities in the hands of disaster planners and city personnel to improve emergency planning and response. The INSIGHT architecture requires a CEP engine that operates on top of a stream processing engine.

The grand challenge, as designed, tests correctness and throughput. We believe, however, that this dataset can serve as a benchmark testing for other aspects of event processing as well. For example, uncertainty management of sensor data and patterns (see a book chapter on the topic [11]) is an intriguing topic that can be tested using this dataset. A task that involves probabilities of analysis such as the *shot-on-goal* query can use this dataset as a benchmark.

An interesting aspect with the high-level description of the data flow graph is the partitioning of data and queries into several threads of execution. For the *TechniBall* setup we chose to divide this into three main parallel computation processes. However, with more CPUs or even additional nodes for computing, this could be massively distributed on a larger set of processors. Finding the most efficient distribution and reasoning about strategies to generally find adequate partitionings of such data flow graphs will be a focus of our future work.

Acknowledgement

We thank Ella Rabinovich for useful discussion. This research has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 318225.

This work has partly been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 *Providing Information by Resource-Constrained Data Analysis*, project A1.

References

[1] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in etalis. *Semantic Web*, 3(4):397–407, 2012.

[2] Arasu et al. Stream: The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[3] A. Artikis, G. Paliouras, F. Portet, and A. Skarlatidis. Logic-based representation, reasoning and machine learning for event recognition. In *DEBS*, pages 282–293. ACM, 2010.

[1] www.insight-ict.eu

[4] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. Moa: Massive online analysis. *The Journal of Machine Learning Research*, 99:1601–1604, 2010.

[5] C. Bockermann. The *streams* framework, 2012. URL `http://www.jwall.org/streams/`.

[6] C. Bockermann and H. Blom. The streams framework. Technical Report 5, TU Dortmund University, 12 2012.

[7] H.-L. Bui. *Survey and Comparison of Event Query Languages Using Practical Examples*. PhD thesis, LMU München, 2009.

[8] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, 1994.

[9] EsperTech. Esper complex event processing engine, 2013. URL `http://esper.codehaus.org/`.

[10] O. Etzion and P. Niblett. *Event Processing in Action*. Manning, 2010.

[11] A. Gal, S. Wasserkrug, and O. Etzion. Event processing over uncertain data. In *Reasoning in Event-Based Distributed Systems*, pages 279–304. Springer, 2011.

[12] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[13] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Comput.*, 4(1):67–95, 1986.

[14] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *NetDB*, 2011.

[15] N. Marz. Storm - distributed and fault-tolerant realtime computation, 2013. `http://www.storm-project.net`.

[16] C. Mutschler, H. Ziekow, and Z. Jerzak. The DEBS 2013 Grand Challenge. In *DEBS*. ACM, 2013.

[17] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, pages 170–177. IEEE, 2010.