DEBS Grand Challenge: Complex Analytics over Sensor Data from Soccer Games

Andreas Harre
andreas.harre@uni-oldenburg.de

Philipp Rudolph
philipp.rudolph@uni-oldenburg.de

Jan Sören Schwarz
jan.soeren.schwarz@uni-oldenburg.de

Dennis Geesen
dennis.geesen@uni-oldenburg.de

Marco Grawunder
marco.grawunder@uni-oldenburg.de

Timo Michelsen
timo.michelsen@uni-oldenburg.de

ABSTRACT

This paper describes our solution for the DEBS'13 Grand Challenge, which deals with the processing of a big amount of sensor data generated during a soccer game. To solve the given problem we used the data stream management framework Odysseus developed at the University of Oldenburg. We present how we used the existing features of Odysseus and which features were added to solve the challenge. Furthermore we describe how we checked the correctness and measured the latency of our solution.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Management

Keywords

Complex Events, Data Stream Processing, Odysseus, Grand Challenge Solution

1. INTRODUCTION

Once again in the context of the DEBS the Grand Challenge was published. The given challenge is to compute a big amount of data created by sensors during a soccer game. The sensors were placed in the shoes of the players, additional in the gloves of the goalkeeper and in the soccer balls. During the 60 minutes of the game the sensors created a data stream with 200 events per player sensor and 2000 events per ball sensor per second. The goal of the Grand Challenge is to process the given data in order to detect events like ball contacts or shots on the goal and to generate statistics over the course of the game.

This year we solved the challenge as a group of three students who wrote their bachelor thesis about topics dealing with the data stream management framework Odysseus [1]. So we decided to use Odysseus as a basis for solving the challenge. Odysseus is designed to build data stream management systems for diverse domains.

The first query analyses the running speed of the players and determines the time a player spent in different speed states (e.g. trot or sprint). The second query computes ball possession statistics for every single player and for both teams. Furthermore the position of every single player on the field over the time is analyzed by implementing a heat map in the third query. The last query details shots on goal.

In section 2 we describe our solution using Odysseus for the different queries in detail. We also explain the visualization of the data in Odysseus before we proceed with the evaluation of our solution in section 3. Finally, we conclude the paper in Section 4.

2. THE SOLUTION

Odysseus was developed to provide an extensible framework for event based systems. It can be used as a base for data stream management systems using relational stream algebra as it provides certain sets of operators and data structures as well as the processing logic. Odysseus itself makes use of Java and the OSGi framework, so it is possible to integrate components, such as operators, in form of packages in order to adjust it to specific requirements. Event based systems can be created using declarative queries which gives the opportunity to easily adjust to small changes during the development of a system. The queries get translated by Odysseus and get executed multi-threaded.

The Procedural Query Language (PQL)[1] is one of the provided query languages. It is not completely declarative and can be used more procedural and functional as for example the SQL-like CQL. An example is shown in the following where a csv based source is defined, some attributes are projected, the sensor events of the players are selected by their sensor id and events containing the current and the last sensor id and timestamp are created by a statemap operator.

```
soccergame := ACCESS({source='soccergame',wrapper='
    GenericPull', transport='file',
  protocol='simplecsv', dataHandler='tuple',
```

[1]http://odysseus.informatik.uni-oldenburg.de

```
  schema=[
  ['sid','integer'],['ts','starttimestamp'],
  ['x','integer'],['y','integer'],['z','integer'],
  ['v','integer'],['a','integer'],
  ['vx','integer'],['vy','integer'],['vz','integer'],
  ['ax','integer'],['ay','integer'],['az','integer']
  ],
  options=[
   ['filename','${DATAFILENAME}'],['Delimiter',',']
  ]
})
extSoccergame = PROJECT(
    {attributes = ['sid','ts','x','y','z','a']},
    soccergame)
player = SELECT(
    {predicate = RelationalPredicate('sid>12␣AND␣sid
        !=105␣AND␣sid!=106')},
    extSoccergame)
lastPlayer = STATEMAP({expressions = ['ts',
    ['__last_1.ts', 'lastTS'], 'sid',
    ['__last_1.sid', 'lastSID']]}, player)
```

We chose PQL for this challenge because it allows us to integrate own operators easily (using annotations in Java) and to use own functions for example to map the running intensity in Query 1.

A framework that already provides the required basic tools allows to focus more on the specific problem. So we were able to spend most of our time on operators, the evaluation and visualizations of solutions to the problems given in the 2013 Grand Challenge. The problem-solving approaches as well as the additions that have been made to Odysseus are explained in the sections about the certain subproblems.

To process big data via data streams these streams have to be connected to Odysseus. Odysseus provides this functionality via the access operator. For solving the challenge we instantiated multiple access operators to handle comma-separated values (csv). The main data for the challenge including the sensor ids, position, speed and acceleration values were given in a well-formed csv file, so we were able to work with this data without problems. Additional metadata including a mapping from sensor ids to player ids and game interruption times were given in not-well-formed files. In our solution we manually transformed these files into three additional well-formed csv files containing the metadata for mapping sensor ids to player ids, for detecting a running game and game interruptions and for recognizing which team attacks which goal depending on the time. For using this metadata in the data stream we enriched the main data with it via the enrich operator which is provided by Odysseus. Accordingly, the processed soccer stream schema contains the sensor id and also the player id to which the sensor id belongs. This purpose simplifies the process of creating the queries because we don't need to do the sensor to player mapping in every single query again [1].

2.1 Query 1 - Running Analysis

In the first query we calculate the running analysis for all players on the field by using the data of all sensors of each player. There are two classes of outputs. One result stream contains the most current running informations including the intensity, distance and speed of the current run. Then there are result streams that contain aggregated running statistics for four different time intervals featuring time and distance values of the six running intensities for each player.

Odysseus automatically generates a query plan of our solution that consists of about 160 operators. So figure 1 is a simplified query plan, that will be explained in the following. At first the input stream is split into one stream per player,

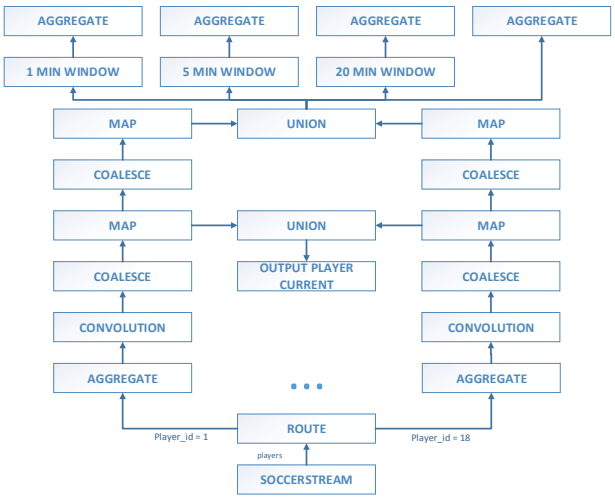

Figure 1: Overview of query 1

so it is possible to detect individual changes in the running intensity of each player later on. In a next step the speed values of the player's sensors get averaged to determine the current player position. For this aggregation we always use the speed values of the four most current events. This has two advantages: First we get a relatively precise approximation of the actual current speed of the player instead of just the last speed value of the last sensor event while still keeping the same frequency and secondly we don't regard outdated sensor data. This could happen if we always used the last known speed values of all the player's sensors, as one sensor could take noticeable longer to update than the others.

In order to take the noise in the raw velocity measurements into account, we use a convolution filter on the resulting events. This is done by using the convolution operator, which filters with a density function, set to use a gaussian distribution and to look at the four values before and the four values after the current value in a density to correct the current speed value. This delays the output stream by ten incoming events, which is tolerable for us due to the high frequency of the incoming events. The noise in the resulting events is now reduced sufficiently, so the following (temporary) running intensity mapping does not fluctuate unreasonably.

The first result stream, containing the current running statistics, is now generated by using an union operator on the resulting stream of each player.

To generate the aggregated running statistics we also use these streams for further analysis. First we use a coalesce operator, which executes aggregations until a given attribute changes. So we aggregate the average running speed on each

stream until the running intensity changes. This way we produce single complex events containing the average speed for distinct runs. Now we map the distance and duration of the run by using the average speed and the timestamp of the last 'run event'. To be able to determine the actual running intensity, we first use a coalesce operator to sum up the distance and time until an event arrives that lies at least 0.8 seconds behind the first regarded event. Then we perform the final running intensity mapping for the current run. To reduce the effort of multiple join operators later, we extend the schema at this point to contain distance and time values for each running intensity by setting the not matching values to zero. Then we merge the streams of all players with an union.

Finally we use this merged stream as an input for aggregations with four different windows with a length of 1, 5, and 20 minutes and the whole game duration. The aggregates sum up the different run times and distances grouped by player.

2.2 Query 2 - Ball Possession

For this query, data about the ball possession of each single players as well as for the two teams has to be computed. Figure 2 depicts a rough overview of our solution.

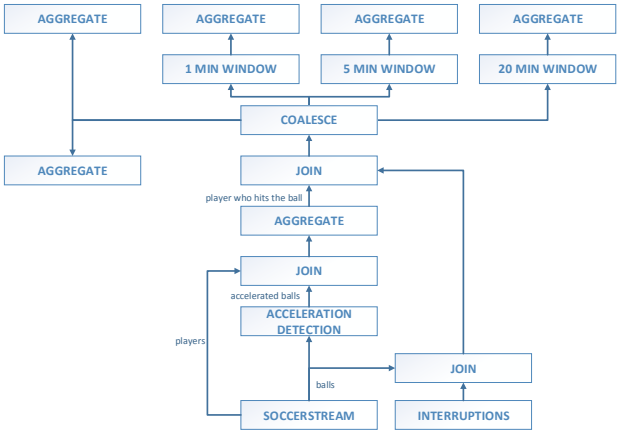

Figure 2: Overview of query 2

To identify ball possession, the first step is to search for ball hits, which were defined as an acceleration of the ball of more than 55 m/s^2 in the problem description. The part of the query summarized in the figure as "acceleration detection" first uses the convolution operator of Odysseus to provide a better data quality. We use a gaussian function with a size of 4. After the convolution the changedetect operator filters out all events despite of the transition between an acceleration of the ball of more and less than 55 m/s^2. So we get a single event when the ball gets into the accelerated state and when it leaves this state again. In the join operator after the acceleration detection the events are joined with the data of every player who is at least in a distance of one meter to the ball. The following minimum aggregation provides the nearest player to whom the hit can be assigned.

As result of the outlined processing we get a single event for every ball hit containing the player id, the team id and the timestamp. These events are joined with a ball stream which previously got enriched with the game interruption metadata events. The metadata events consists of a start and an end timestamp and the current state of the game as a boolean value, which provides a high flexibility. So we have a stream with all ball events and additional information about the game status and the player and team who possess the ball at the moment. Via a coalesce operator we aggregate this stream grouped by the player id and the status of the game. This way we get a single event for every ball possession containing information about the player and the duration of the ball possession. With this information we can generate the first demanded result with an aggregation grouped by the player id. This result stream sends events containing the player id, the total duration of his ball possessions and the total count of his ball possessions.

For the second requested result we aggregate the ball possession events grouped by team id with different window sizes. Due to the requested sliding windows for the output it is not easy to compute the ball possession of the team exact for the given time periods of 1, 5 and 20 minutes. Because the ball possession events do not exactly fit in the windows, the percentaged ball possession can not be computed via division by the window length. To get a reasonable result the total ball possession time of both teams in the window is computed. With this value it is possible to compute a reasonable percentaged ball possession.

2.3 Query 3 - Heat Map

The third query aims to calculate the relative time a player spend in a specified cell of the field in consideration to a specified time window. The parameters, for calculating the cells on the field and also the size of the time window which has to be considered, are given in the task description. Figure 3 gives an simplified overview of the query plan of this query. The automatically generated query plan contains around 850 operators which is caused by considering the different players and the different time windows.

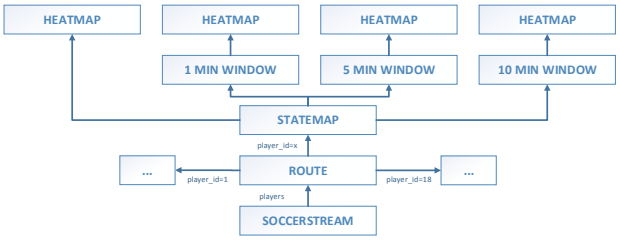

Figure 3: Overview of query 3

The first task is to convert the sensor id of the input stream to the player id. So the position of a player is the result of the position of every sensor which belongs to the player. The resulting stream is processed by a route operator which splits the stream into multiple substreams depending on a route-predicate. The predicate aims to split the stream into substreams depending on the player id. Every substream contains the events of a specified player. The next step is

to calculate the time a player spent at a position on the game field. The statemap operator takes the last position and the actual position into consideration. The difference of the starttimestamp of the actual position and the start-timestamp of the last position calculates the time the player spent on the last position. After this calculation the stream is piped into the 4 different window operators specified in the task description.

Up to this point Odysseus didn't contain a standard operator for calculating a heatmap. Based on the extensibility of Odysseus, we were able to simply implement a new operator. For the calculation of the absolute time a player spends in a specified cell we developed a heatmap operator which considers the number of rows and the length of the x and the y-axis of the field and also the time a player spent at a position. Based on the number of rows and the length of the axis the operator internally calculates the different cells of the field. The position of an incoming event is mapped to a calculated cell. Additionally every calculated cell contains a time value which represents the time a player spent at a cell. For this calculation the events in the preceding time window are considered. New events increase the time value of the specified cell and outdated events decrease.

First we implemented the heatmap operator according to the problem description sending a single event for every cell. The resulting stream contained the timestamp, the player id, the x- and y-coordinates of the lower left and the upper right corner of the cell and the calculated time spent at this cell. The total time had to be converted to a percentaged value in a following map operator. But due to the high data output of 34016 tuples per second per player the processing was very slow. So we decided to merge the events of the different cells into one large event with a lot of attributes. Also we changed the total time output value to a percentaged value to remove the additional map operator.

For evaluating the result of this query we developed a heatmap visualization which is introduced in section 2.5.

2.4 Query 4 - Shot on Goal

The solution for the fourth query is partly based on the solution for query 2 (s.a. section 2.2) and is shown as an overview in figure 4.

The first part delivers similar to query 2 an acceleration state and the nearest player at the moment of the ball hit. To detect shots on the goal the complete ball stream is joined with the previous generated events. So we get a stream containing all ball events enriched with the acceleration state and information about the current player. Afterwards a statemap operator is used to get the player id of the last event and the following coalesce operator aggregates the stream according to the player id and the current distance of the ball and the position the shot took place. So we get pairs of the first and last position and timestamp within an area of one meter away of the position the shot took place.

Because the task was to detect a shot within one meter, it is checked at this point in the query for the first time if the ball would strike the goal. To simplify the complexity of the PQL code we implemented new functions for this. Odysseus already offers spatial functionality using the JTS[2] framework. First we decided to use this existing functionality, although it only provides 2D functionality, because the complexity of the solution is not specified in the challenge description and we decided to set our focus on the other problems. So we just had to use the Odysseus function provider to add functions, which created a geometry object out of the given coordinates and detected shots on a goal. Our shot on goal detection function created a spatial line and checked if it intersects with an constant polygon representing the goal area.

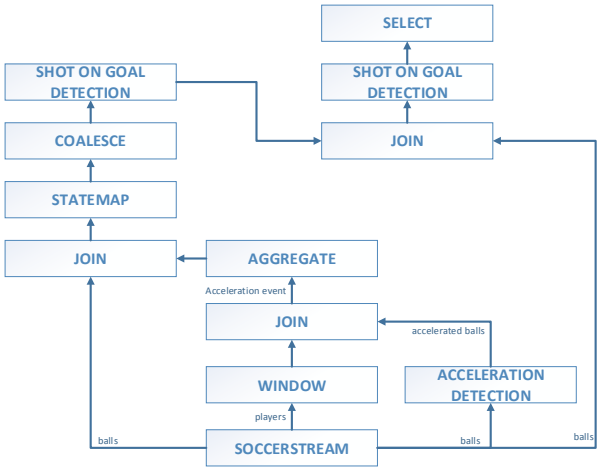

Figure 4: Overview of query 4

But during evaluation we found out, that the numerous generation of new objects seems to cause a high decrease of performance. So we implemented a new "ShotOnGoal" function, which uses simplified geometric computation to increase the throughput of data. Our implementation does not involve the sensors acceleration and speed data. The speed is computed by the delta of the position and the timestamp. We simplify the processing and assume that the speed is constant during the shot and check if the ball would strike the goal area during the next 1.5 seconds.

The result of the detection function is provided as an integer attribute in our events. The value -1 stands for a shot on the left goal and 1 for the right goal, while 0 represents no shot on a goal. We created a further metadata stream to check if the shot will strike the opponents goal to filter out shot on the own goal. Finally we join the events again with the complete ball stream, check once again for a shot on the goal and filter out all events that are no shot on a goal. This is done to identify the end of the shot on the goal and to get the demanded output data.

2.5 Visualization

Odysseus provides a customizable graphical user interface which is based on the Eclipse Rich Client Platform (RCP) called Odysseus Studio 2.

[2]http://www.vividsolutions.com/jts/JTSHome.htm

Our solution uses Odysseus Studio 2 for data stream visualization. Odysseus Studio 2 supports different visualization types such as a list view for visualizing data and the corresponding metadata (e.g. start- and endtimestamp) or a table view for visualizing data and corresponding column names. It is possible to visualize the output of every single operator used in the query plan simultaneously to different views.

As a part of our solution we developed an abstract view called "soccer view" for visualizing the given soccer data and also for evaluating the result of our defined queries. The "soccer view" is an on-line visualization so it is directly executed and controlled via incoming data stream elements. It makes intensive use of the Standard Widget Toolkit (SWT)[3] and its drawing capabilities to build up the visualization according to our suggestions. This view initializes an image of a soccer field as a background image. Also this view provides coordinate transformation methods to transform given positions in the data stream to calculated position on the soccer field background.

The "soccer view" is instantiated by two different views, the "position" visualization and the "heatmap" visualization. Both views show the current timestamp of the last data stream element, the resulting milliseconds after the start of the game and the formatted time in minutes, seconds and milliseconds. This time information helps to determine the current time of visualized game situation. The information is positioned in the upper left corner of every soccer view.

The position visualization as its name implies visualizes the correct position of every single player, the referee and the ball. Being more precisely this view shows the correct position of every single sensor. The position of a sensor is shown as a small color filled circle. The visualization uses different colors for the different type of game participants. Players of the team 1 are red colored, players of the team 2 are blue colored, the ball is yellow colored and the participating referee is black colored. For recognizing the different participants in the view we also added the player id next to the sensor representing the left leg of the participant respectively the sensor id of the ball as a text in the visualization. This view can be used to analyze the game situation according to a specific point in time of the soccer game. Figure 5 shows the position visualization at the point in time 87 seconds after the game has started.

The heatmap visualization is a concrete visualization for evaluating the results of query 3 presented in section 2.3. This visualization displays in which cells of the field a specific player has spent time by using a graduated color scheme. Cells in which the player spent more time are colored rather red and cells in which the player spent less time are colored rather yellow. Cells in which the player spent no time are not visualized. The heat map visualization makes big profit of the schema of the resulting data stream which includes the downer left and upper right corner of the cell and the relative value of time spent in this cell. The relative value of time is mapped to a specific color. Figure 6 shows the heatmap visualization for player with id 15 at the point in time 29 minutes after the game has started on the game field which split in 25x16-cells.

[3]http://www.eclipse.org/swt/

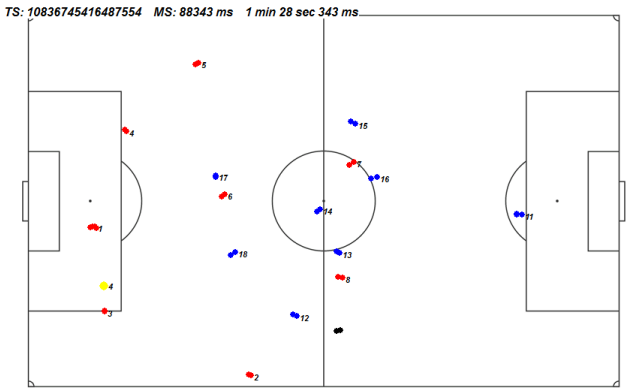

Figure 5: position visualization

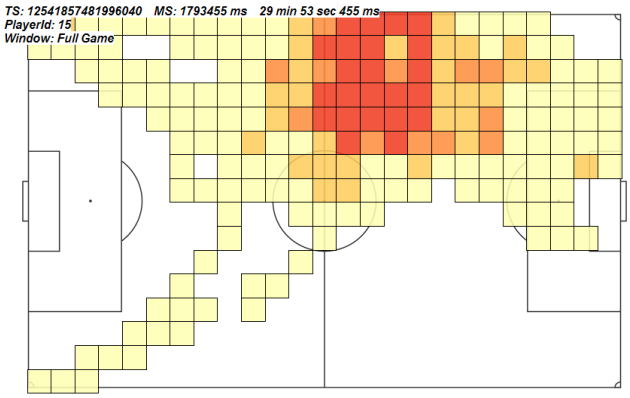

Figure 6: heatmap visualization

An important design decision for the architecture of the soccer view is to decouple the thread which receives and processes the data stream elements and the thread which draws the visualization. A bad solution is to redraw the visualization every time a new data stream element is received. The data rate is too high for this purpose so the system would permanently redraw the visualization. So in our solution we decided to cache incoming events and to redraw these cached events asynchronously after a fixed time interval (100 milliseconds in case of the positions view an one second in case of the heatmap view). Because of the usage of Odysseus Studio 2 we were able to implement the additional complete soccer visualization within around 400 lines of code.

3. EVALUATION

To evaluate our solution, we checked the correctness of our results as well as we measured the throughput and latency of our system. We made different experiments to determine latency and throughput of the queries. We run the experiments on an AMD FX 8350 with 4 GHz and 32 GB of main memory. The Java virtual machine for Odysseus Studio got 16 GB of main memory. In the first experiments we wanted to determine what are the throughputs of the dif-

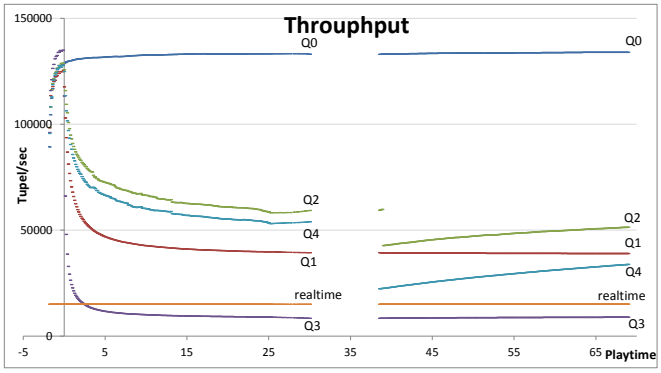

Figure 7: Throughput in Tuple/sec

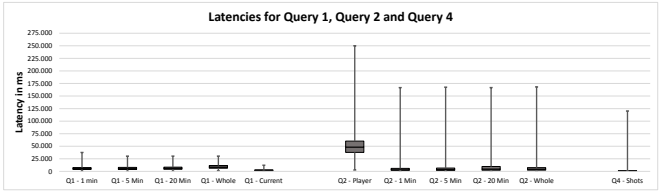

Figure 8: Latencies for query 1,2 and 4

ferent queries. Figure 7 shows the tuples that the systems processes per second for each query. The x-axis shows the corresponding play time (which we think is more informativ than the number of tuples processed until that time).

For that, we initially measured how many tuple per seconds the system can process. Process means in this case: Read vom csv and write as csv. In the diagram in Fig. 7 this is the curve Q0. As you can see the system can roughly read and write 130 000 tuple per second, which is much higher than the required input data rate of 15 000 tuples (stated as realtime). The diagram shows also that all queries besides query 3 (heatmap) can process the data much faster, than needed.

Obviously, our system had some problems with the missing ball to the end of the last 2 minutes of the fist half. Query 2 (ball possession) and query 4 (shots on qoal) broke down dramatically. We had no time to make further investigations, but be think the problem is, that the system needs to keep more sensor data of players in memory that could possibly be merged with a ball measurement. We need to further investigate if the problem is still there when we start a new query processing to each half of the game.

We validated the correctness of our results by using multiple resources. First the used the manually created ball possession statistics and shots on goal statistics as well as the video recording of the game. Both of these have been provided provided to us as reference data. Then we also used our own visualizations, which are described in chapter 2.5. Moreover we compared these resources with our generated results of the different queries by doing additional computations.

4. CONCLUSION

For our solution of the DEBS 2013 Grand Challenge we used the data stream management framework Odysseus. The queries were expressed by means of the Procedural Query Language (PQL) of Odysseus. Due to the extensibility of PQL we were able to implement new operators and functions for parts that were not expressible in PQL until then.

To optimize performance of processing the computation could be shared over more nodes by adding annotations. But in the problem description the computation in one VM was requested. Anyway as described in section 3 our solution is able to process the queries in real time.

5. REFERENCES

[1] H.-J. Appelrath, D. Geesen, M. Grawunder, T. Michelsen, and D. Nicklas. Odysseus: a highly customizable framework for creating efficient event stream management systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, 2012.