Grand Challenge: Implementation by Frequently Emitting Parallel Windows and User-Defined Aggregate Functions

Sobhan Badiozamany
Uppsala University
Sweden

Lars Melander
Uppsala University
Sweden

Thanh Truong
Uppsala University
Sweden

Cheng Xu
Uppsala University
Sweden

Tore Risch
Uppsala University
Sweden

Emails: Firstname.Lastname@it.uu.se

ABSTRACT
Our implementation of the DEBS 2013 Challenge is based on a scalable, parallel, and extensible DSMS, which is capable of processing general continuous queries over high volume data streams with low delays. A mechanism to provide user defined incremental aggregate functions over sliding windows of data streams provide real-time processing by emitting results continuously with low delays. To further eliminate delays caused by time critical operations, the system is extensible so that functions can be easily written in some external programming language. The query language provides user defined parallelization primitives where the user can express queries specifying how high volume data streams are split and reduced into lower volume parallel data streams. This enables expensive queries over data streams to be executed in parallel based on application knowledge. Our OS-independent implementation was tested on several computers and achieves the real-time requirement of the challenge on a regular PC.

Categories and Subject Descriptors
H.2.4 [**Database Management**]: Systems – *Parallel databases*, *Query processing*

Keywords
Parallel data stream processing; continuous queries; spatio-temporal window operators.

1. INTRODUCTION
Monitoring a soccer game requires a system than can process, in real-time, large volumes of data to dynamically determine physical properties as they appear. This requires a system having the following properties:

- To keep up with the very high data flow the system must deliver high throughput while processing expensive computations over high volume data.
- Response in real-time requires continuous delivery of query results with low latency.
- Continuous identification of physical phenomena, such as moving balls and players, requires complex spatio-temporal algebraic computations over windows.

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

Our EPIC (Extensible, Parallel, Incremental, and Continuous) DSMS provides very high throughput and low latency through parallelization, extensibility, and user defined incremental aggregation of windowed data streams. The high level query language provides numerical data representations and data stream windows as first class objects, which simplifies complex numerical computations over streaming data and enables automatic query optimization. To provide very high performance of low level numerical and byte processing functions the system is easily extensible with user defined functions over streams and numerical data, which allows accessing external systems and plugging in time-critical user algorithms.

EPIC extends the SCSQ system [9] with several kinds of data stream windows and incremental evaluation of user-defined aggregate functions over the windows. In particular the window operator *FEW* (Frequently Emitting Windowizer) decouples the frequency of emitted tuples from a window's slide.

To process expensive queries with high-throughput and low latency the system provides application specific stream parallelization functions where general *distribution queries* specify how to parallelize and reduce outgoing data streams.

2. THE EPIC APPROACH
First FEW and its incremental user-define aggregation are presented in sections 2.1 and 2.2, and then the solution is outlined in section 2.3.

Figure 1 shows the overall data stream flow of the implementation. The thickness of the arrows in all data flow diagrams in this paper correspond to the relative volume of the data streams. Each node in the dataflow diagram is a separate OS process, called a *query processing node,* in which a partial continuous execution plan is running. The topology of the dataflow diagram is completely expressed in the query language where it is possible to specify continuous sub-queries running in parallel [9]. The system automatically creates OS processes running the execution plans of the sub-queries and the communication channels between them (local TCP). In the Grand Challenge implementation, the query processing nodes all run on the same computer and the OS is responsible for assigning CPUs to the processes. The system can also distribute query processing nodes over several computers but those features are not used here.

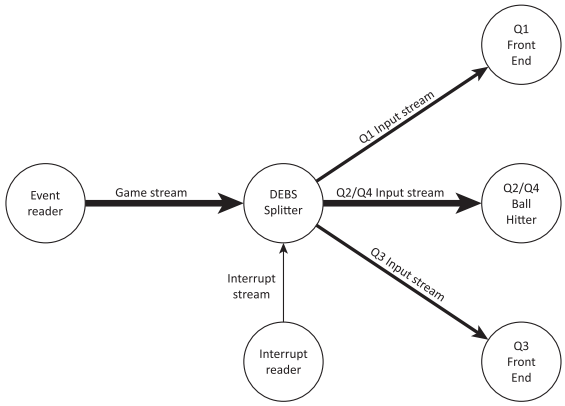

Figure 1. High level data stream flow

2.1 Frequently Emitting Windowizer, FEW

EPIC provides window forming operators that support several kinds of windows, including time, count, and predicate windows [5][2][7]. The windows are formed by *window functions* mapping streams to streams of objects of type *Window*. For example, the window function

tWindowize(Stream s, Number length, Number stride) -> Stream of Window ws

forms a stream *ws* of timed windows over a stream *s* where windows of *length* time units (seconds) slide every *stride* time units. To avoid copying, the windows are represented by pointers to their first and last elements. When a window slides the pointers are updated.

A naive implementation of *tWindowize()* would emit tuples only when the formed windows slide. This causes substantial delays, in particular for large windows. For example, when forming a 10 minutes window, it is not practical to wait 10 minutes for the aggregation to be emitted. To be able to emit aggregation results before a complete window is formed, we have introduced a window function having a parameter *ef*, the *emit frequency*:

fewtWindowize(Stream s, Number length, Number stride, Number ef) -> Stream of Window pw

The window forming function *fewtWindowize()* forms partial time windows*, pw,* every *ef* time units. The emitted partial windows are landmark sub-windows of the elements of the window being formed. When the formed window is complete it is emitted as well before it slides, and then the landmark is reset to the start time of the newly slided window.

The FEW windows are required when:

- The results must be emitted before the window is formed.
- The results must be emitted more often than the slide (not used in this application).

2.2 User-defined incremental window aggregate functions

The windowing mechanism in EPIC supports incrementally evaluated user defined aggregate functions [1][8]. These are defined by associating *init()*, *add(),* and *remove()* functions with a user defined aggregate function:

- *init() -> Object o_new* creates a new *aggregation object, o_new*, which is used for accumulating changes in a window.
- *add(Object o_cur, Object e) -> Object o_nxt* takes the current aggregation object *o_cur* and the current stream element *e* and returns the updated aggregation object *o_nxt.*
- *remove(Object o_cur, Object e_exp) -> Object o_nxt* removes from the current aggregation object *o_cur* the contribution of an element *e_exp* that has expired from a window. It returns the updated *o_nxt*.

A user defined aggregate function is registered with the system function:
aggregate_function(Charstring agg_name, Charstring initfn, Charstring addfn, Charstring removefn) -> Object

For example, the following shows how to define the aggregate function *mysum()* over windows of numbers:

```
create function initsum() -> Number s as 0;

create function addsum(Number s_cur, Number
e) -> Number s_nxt as res + e;

create function removesum(Number s_cur,
Number e_exp) -> Number s_nxt as s_cur –
e_exp;
```

These functions are registered to the system as the aggregate function *mysum()* by the function call:

```
aggregate_function('mysum','initsum','addsum
','removesum');
```

After the registration *mysum()* can be used in CQs as:

```
select mysum(w) from Window w where w in
fewtWindowize(s, 10, 2, 1);
```

In this simple example the aggregation object is a single number. It can also be arbitrary objects, including dictionaries (temporary tables) holding sets of rows, which is used in the Challenge implementation to incrementally maintain complex spatio-temporal aggregations.

2.3 Solution outline

In Figure 1 the *Event Reader* node reads the full-game CSV file and produces the *Game* stream consisting of events for both balls and players. The *Event Reader* then scales the time stamps by subtracting the start time. It also transforms the position, velocity, and acceleration values to metric scales. To avoid the *Event Reader* becoming a bottleneck it is implemented as a foreign function in C. To speed up the communication we use binary representation of all events communicated between query processing nodes, while the input and output log files use the CSV format.

The *Interrupt Reader* node produces the *Interrupt* stream, which contains referee interruptions, by reading and transforming the provided game interruptions files.

The *DEBS Splitter* node merges the two input streams based on the time stamps in the streams and produces parallel input streams for the different queries. It also filters out those event stream tuples of the *Game* stream that are in-between game interruptions. The nodes *Q1 Front End*, *Q2/Q4 Ball Hitter*, and *Q3 Front End* receive parallel data streams required for the four Grand Challenge queries Q1-Q4. Q2 and Q4 share some downstream computations executed by *Q2/Q4 Ball Hitter node*.

In EPIC the *splitstream()* system function provides customizable distribution and transformation of stream tuples. The user can provide customizable splitting logic as a *distribution query* over an incoming tuple that specifies how a tuple is to be distributed, filtered and transformed.

The distribution query for the *DEBS Splitter* in Listing 1 is passed as an argument to *splitstream()*.

```
1 select i, ev from Integer i
2 where (i = 0 and isPlayer(ev)) or
3       (i = 1) or
4       (i = 2 and isPlayer(ev));
```

Listing 1. DEBS Splitter distribution query

The result of the query are pairs *(i, ev)* specifying that an incoming event *ev* is to be sent to output stream number *i*. In the DEBS splitter distribution query three output streams enumerated by *i* are specified. They produce the corresponding streams *Q1 Input*, *Q2/Q4 Input*, and *Q3 Input*. The Boolean function *isPlayer(v)* returns true if *v* is a player sensor reading.

To speed up the processing, shared computations are made in separate nodes. In Figure 1 the *Q1 Front End* and the *Q3 Front End* nodes perform stream preprocessing and reduction for queries 1 and 3, respectively, while the *Q2/Q4 Ball Hitter* node detects hits to the ball needed by queries 2 and 4.

2.3.1 Query Q1: Running Analysis

Figure 2 shows the topology of Q1. The aggregated running statistics for different time windows are computed in parallel based on the common current running statistics produced by the *Q1 Front End* node. The stream containing player sensor readings is sent to the *Q1 Front End* node (see Listing 1), which produces the running statistics. The running statistics is then broadcasted to four other nodes to compute the aggregated running statistics of different time window lengths.

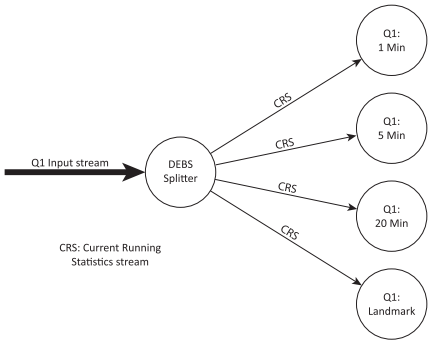

Figure 2. Query 1 data stream flow

2.3.1.1 Incremental maintenance of running statistics

In order to make the result more reliable for the current running statistics, we first create a *1 s* tumbling window and then calculate the statistics for each player over that window. The window length *1 s* was chosen experimentally to produce stable results. Both running and aggregate statistics utilize user defined aggregate functions to maintain arrays of the two types of statistics for each player.

2.3.1.2 Current running statistics

For each incoming player sensor reading in the current *1 s* window, the following statistics tuple for each player is incrementally maintained in an array:

(ts_start, ts_stop, pid, left_x_start, left_y_start, left_x_stop, left_y_stop, right_x_start, right_y_start, right_y_stop, right_y_stop, sum_speed, count)

The time stamp *ts_start* stores the first time when a sensor reading of player *pid* arrives to the current window, while *ts_stop* stores the last sensor reading. The elements *left_x_start*, *left_y_start*, *right_x_start,* and *right_y_start* are the position readings of the left and right foot of the player at time *ts_start*, while *left_x_stop*, *left_y_stop, right_x_stop,* and *right_y_stop* are the corresponding foot position readings at time *ts_stop*. To incrementally calculate the average velocity the elements *sum_speed* and *count* are also included. *ts_start*, *left_x_start*, *left_y_start*, *right_x_start,* and *right_y_start* are updated only when the first sensor reading of the player *pid* arrives to the window, while all the other elements are updated every time a sensor reading of *pid* arrives. Here, no remove function is needed for the aggregation, since we are maintaining a stream of tumbling windows where the statistic will be re-initialized every time the window tumbles.

With the statistics above, the current running statistics for a given player is calculated as the Euclidian distance between the average position of the first and last update during the time window.

2.3.1.3 Aggregate running statistics

We have chosen to log the result tuple of Q1 in CSV format every *1 s* since the current running statistics are not emitted more often than once per second. Four FEW time windows were defined for aggregating running statistics with lengths 1 minute, 5 minutes, 20 minutes, and the entire game. All windows slide and emit results every *1 s*. FEW is critical for early emission while the first windows are being formed.

Aggregate running statistics over the window are incrementally maintained in an array similar to current running statistics.

The stream from the *Q1 Front End* node contains the elements *ts_start, ts_stop, player_id, intensity, distance,* and *speed*. The difference *ts_stop – ts_start* is used to incrementally maintain the duration of a player being in the corresponding running *intensity* class. Analogously, the moving distance is maintained for the corresponding intensity classes by incrementally associating the incoming distance with the right intensity.

2.3.2 Query Q2: Ball Possession

Figure 3 shows the data flow of queries Q2 and Q4 combined. The *Q2/Q4 input* stream consists of player, ball, and interrupt sensor readings. The *Q2/Q4 Ball Hitter* computes the *Ball Hitter* and the *Ball* streams. The *Ball Hitter* stream contains ball hitter events, which occur when a player *pid* at timestamp *ts* hits the ball. The *Ball* stream contains *Ball Hitter* events interleaved with ball sensor readings. The *Q2/Q4 Ball Hitter* node emits the *Ball* stream to the *Shot on Goals* query processing node, which executes the final stages of query Q4. The *Ball Hitter* stream contains only ball hitter events and is sent to the *Player Possession* node, which calculates and broadcasts the same *Player Ball Possession* stream to four *Team Possession* query processing nodes. The *Team Possession* nodes log every *10 s* statistics of team ball possessions for the two teams with the different window lengths: 1 minute, 5 minutes, 20 minutes, and a landmark window of the entire game. As an alternative, we also measured reporting

team possessions every *1 s* resulting in the same latency and throughput.

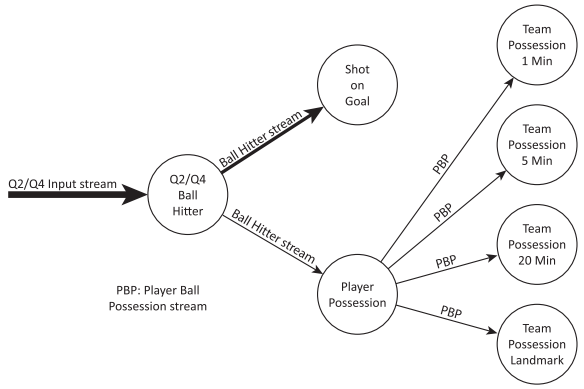

Figure 3. Query 2 and Query 4 data stream flow

2.3.2.1 The Q2/Q4 Ball Hitter query processing node

In order to compute a stream of ball hitters, we maintain acceleration of the ball *ballacc*, its position *bx, by, bz*, the shortest distance from a player to the ball *sdist*, and the player *pid*.

For every input ball sensor reading, the *Q2/Q4 Ball Hitter* node incrementally updates the ball acceleration and the ball position accordingly. When a player sensor reading arrives, it incrementally maintains *sdist.*

A ball hitter event is emitted when both the following criteria hold:

- C1: The ball acceleration reaches a predefined threshold: $ballacc > 55\ m / s^2$.
- C2: The shortest distance *sdist* is within the player's proximity: $sdist < 1\ m$.

There are 36*200 player sensor readings per second. In addition, after being hit, the ball acceleration remains high for a while, in particular before the ball leaves the player's proximity. Therefore, the two conditions C1 and C2 will hold for a short period of time within which several ball hitter events could be reported for the same actual ball hit by the player. To avoid generating false *ball hitter* events, we employ a *dropping policy* to drop player sensor readings occurring significantly later than the last report time. The dropping policy is expressed by the following query condition over a player sensor reading *v*:

```
ts(v)-lrts > epsilon;
```

Here, *lrts* is the latest timestamp when a ball hitter event was reported, and *epsilon* is the minimum time period between two reports. Because Q4 is more sensitive to the *ball hitter* events, we have empirically tuned this parameter to *0.2 s* to get the best possible accuracy of Q4.

2.3.2.2 The Player Possession query processing node

The *Player Possession* node emits the *Player Ball Possession (PBP)* stream consisting of the variables *fts*, *pid*, and *hits*, which state that the player *pid* possessed the ball *hits* times, starting from first time the player hits the ball, *fts.*

The *Player Possession* node increases the variable *hits* if a ball hitter event *bhe* is from the same player *pid*. Otherwise, it will emit ball possession events for player *pid* and then reset the variables. The total possession time is the interval between the timestamps *bhe* and *fts*.

2.3.2.3 The Team Possession query processing nodes

There are four *Team Possession* nodes, each with different window length: 1 minute, 5 minutes, 20 minutes, and a landmark of the whole game. For the received *Player Ball Possession* stream they compute team possession statistics as follows:

- Incrementally calculate the sum of the ball possessions of all players in each team when a corresponding player ball possession arrives.
- When a report is logged, the following two percentages are calculated:

$$P_A = \frac{sumTeamA}{sumTeam\ A + sum\ TeamB}$$

$$P_B = \frac{sumTeamB}{sumTeam\ A + sum\ TeamB}$$

Here FEW windows are used to frequently report while the first windows are being formed. For example, the results must be regularly delivered every *10 s* while the *team possession landmark* window is being formed.

2.3.3 Query 4: Shot on Goal

The *Shot on Goal* node receives three different kinds of events in the *Ball* stream:

- A ball hitter event marks a shot and contains a time stamp and the *pid* of the shooting player.
- A ball event contains the current ball sensor reading.
- An interrupt event indicates a game interruption. It is good practice to reset the shot detection when an interruption occurs.

Q4 shares detection of a ball hit with Q2. However, the logic for detecting a shot is slightly different for the two queries: Q2 is specified stricter than needed for Q4. To share computations this stricter logic is also used for Q4.

The operation of Q4 is straightforward; it is an iteration over the *Ball* stream to keep track of the state of a shot:

1. Wait for the next ball hitter event.
2. Check ball events until the ball has travelled one meter.
3. Return ball events as long as the ball is approaching the opposite team's goal.

The calculation of the ball direction uses basic linear algebra over the ball sensor readings.

Gravity is accounted for to an extent. The expected time for the ball to travel to the goal line is multiplied twice with the acceleration constant *g*, and added to the height of the goal bar. The actual ball trajectory is not considered, but the current calculation should be an adequate approximation.

Using the Q2 requirements for detecting a ball hit has the drawback that some events are not detected, such as the header at 12:19 in the second half our example *Game* stream, since the ball is more than one meter away from any sensor. Whether that is technically a "shot" is questionable.

Curve balls need special attention. For example, at 26:07 in the first half there is a curve ball goal. In this case the direction of the ball is pointing outside the goal posts, while the ball later curves inwards and comes to rest inside the goal.

To handle curve balls we have introduced a state *pending*, indicating that a shot is not yet dismissed, but could later be

become a shot on goal. The model adds two meters of margin on both sides of the goal posts and the shot is considered pending if it points in the direction of the margin area.

Bounces are considered as long as the direction of the bounce is within the negative distance of the goal bar plus gravity. While the instructions do not account for bounces at all, this limit should add some correctness to the algebra.

Shots that are bounces, which we detect, are not included in the provided list of shots on goal. In the second half of the game there are four shots on goal that are bounces. They are at 4:11, 19:39, 24:36 and 29:29. Setting the bounce threshold to zero, i.e. not considering bounces creates a result in accordance to the specification. Viewing the video makes it apparent that the specification is not correct in this regard.

2.3.4 Query 3: Heat Map

In Query 3 a grid on the field is formed where the cells are numbered in row order, for example from 0 to 6399 in a 64 X 100 grid. Given the position of a player *(x,y),* the function *cell_id(x,y,grid_size)* returns the corresponding cell number for a given grid size. Query results for lower resolution grids are computed by aggregating the results for the higher resolution grids. Thus we incrementally maintain the results only for the highest resolution.

Note that the results of longer windows cannot be built on top of the results from a shorter window. This is due to the *1 s* stride parameter in all the queries. For example, the 5 minute window can't be built on top of the results produced by the 1 minute window, since the 5 minute window needs to remove the contributions made to the statistics by the expired elements, i.e. the elements with the time stamp *ts – 300 s*, where *ts* is the current time stamp. Those elements are too old to be in the 1 minute window. Nevertheless, the definition of longer windows in terms of shorter ones could have been utilized if the stride was one minute instead of the one second stride in the Challenge specification.

2.3.4.1 Q3 Front End

Figure 4 shows the dataflow diagram for query Q3. As specified in Listing 1 the *Q3 Input Stream* contains all player sensor readings. The *Q3 Front End* node produces the *One Second HeatMap (OSHM)* stream by forming *1 s* tumbling windows over the incoming tuples. Thereby incremental user defined aggregate functions are used to maintain statistics per second in a table *heamap1s(pid, cell_id, ts, cnt)* local per window. Here *ts* is the latest time stamp player *pid* has been present in the cell identified by *cell_id* cell identifier in the highest resolution grid (64 X 100). *cnt* is the total number of sensor readings for player *pid* in the cell in the current window.

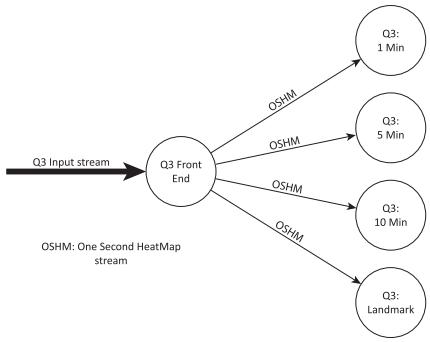

Figure 4. Query 3 data stream flow

The *OSHM* stream is produced by emitting all the rows accumulated in the table during the past second.

The *Q3 Front End* significantly reduces the stream volume by summarizing it. It receives 200 tuples per second from 36 sensors, in total 7200 tuples/second. It emits at maximum the total number of cells all the players have been present in the highest grid resolution during one second, which is about 70 tuples per second, i.e. a factor 10 reduction in stream flow.

2.3.4.2 Q3 query nodes

The *OSHM* stream is broadcasted to four Q3 query nodes *Q3 1 Min*, *Q3 5 Min*, *Q3 10 Min*, and *Q3 Landmark*. These nodes run parallel CQs over time windows with lengths 1, 5, 10 minutes, and whole game, respectively. The windows are formed by the FEW window specification *fewtWindowize(oshm, length, 1, 1),* where *length* is 60s, 300s, 600s and the whole game duration, respectively. The stride and the emit frequency are both *1 s*. The emit frequency is needed so that sub-windows are emitted while the window is being formed the first time.

Similar to *Q3 Front End*, the Q3 query nodes incrementally maintain user defined aggregates by updating the following local tables inside each window as the input stream elements arrive:

heatmap(*pid, cell_id, ts, cnt*)

sensor_count(pid, total_cnt)

In table *heatmap*, the attribute *cell_id* is the cell player *pid* has been present in, *ts* is the latest time player *pid* was in the cell, *cnt* is the number of times the player has been present in the cell. To enable translation of *cnt* into percentages per cell, the Q3 query nodes also maintain *total_cnt* per player, which stores the total number of position reports in all cells for a given player during the window in question.

Since Q3 query nodes only maintain the statistics for the highest resolution in a given window length, at reporting time they compute lower resolutions by aggregating grid cells per player to fill the bigger cells in the higher resolutions.

The Q3 query nodes log the output CSV streams to files. Since each Q3 query nodes cover all grid settings in a given window size, the produced log files contains output stream elements for more than one grid setting. We use the following grid identifiers to tag streams per grid: *6400* for 64 X 100, *1600* for 32 X 50, *400* for 16*25, and *104* for 8 X 13 grid setting.

The size of these log files is huge (ca 400,000 rows/s) since they cover all movements between grid cells over several very long windows. Here it becomes important to use SSD as storage medium, which is fast at writing big blocks in parallel, while disk arm movements for writing different log files has been observed to slow down the entire system throughput with a factor of around two.

3. PERFORMANCE

We measured the performance of our implementation based on both throughput and delay. The throughput was measured as the total execution time per query and for all queries in parallel over the entire game. The latency was measured by propagating the system wall clock of the entry time of the latest event contributing to each result tuple. The delay was calculated by subtracting the propagated entry time from the wall time when a result tuple is delivered. The throughput is measured per query while the latency is measured per output stream.

We ran our experiments on a VMware virtual machine with Windows Server 2008 R2 x64, running on a laptop with the

following specifications: Dell Latitude E6530, CPU: Intel Core i7-3720QM @2.60 GHz, RAM: 8 GB, Hard Disk Device: ST500LX003-1AC15G, OS: Windows 7 64-bit.

Figure 5 illustrates the throughput of the individual queries as well as all queries running together. Queries Q1, Q2, and Q4 take around 5 minutes to finish separately, while Q3 takes considerably longer time, which is mainly due to intensive report computations in the Q3 query nodes. To investigate the log writing time, *Q3* and the *all queries* columns have a watermark indicating how much time it takes to execute them without logging to disk, showing that this takes around 35 % of the Q3 alone time and 25 % of all queries together. We also investigated whether it would be favorable to parallelize the logging of the result stream for Q3 query nodes, but that turned out to be slower in our current environment.

Since all queries run in parallel according to the dataflow diagrams, running all of them together takes approximately the same time as running the slowest one, Q3.

Figure 6 shows the average delay per output stream while running all queries together. Notice that Q2 and Q4 are time critical queries since they immediately report real-time phenomena. By contrast Q1 and Q3 report delayed statistics aggregated over time.

The VMware virtual machine containing our implementation of the Grand Challenge can be downloaded from http://udbl2.it.uu.se/DEBS/. There is also a zip archive that can be run on any Windows machine.

4. RELATED WORK

In the stream processing community, there has been a lot of work for developing query languages over data streams [5]. [7] introduced a formal specification of different kinds of windows over data streams and provided a taxonomy of window variants. The notation of report (emit) frequency was proposed in SECRET [2] without any actual implementation. SECRET is a descriptive model to help users understand the result of window-based queries from different stream processing engines. Esper [4] also allows a report frequency but does not have user defined window aggregate functions. Furthermore Esper's sliding window model is different from FEW because the slides are triggered by window content changes rather than explicitly specified time periods.

To efficiently calculate the aggregate result over long windows with small strides, [6] and [1] use delta computations to reduce the latency and the memory usage. The focus of [8] is to extend a DSMS with online data mining facilities by user defined aggregate functions over windows. The implementation described in this paper shows that EPIC is general enough to define very complicated user defined aggregations as functions while in [1] and [8] the aggregates are defined as updates.

5. CONCLUSIONS

We have addressed the Grand Challenge by expressing continuous queries in a high level language that supports incremental evaluation of aggregate functions over windows and frequently emitting windowing. We meet the real-time requirements of the real-time queries on a virtual machine running on a laptop. The extensibility of the query engine was used for supporting high throughput and low latency of time critical operations.

ACKNOWLEDGMENTS

This work was supported by the Swedish Foundation for Strategic Research, grant RIT08-0041 and by the EU FP7 project Smart Vortex.

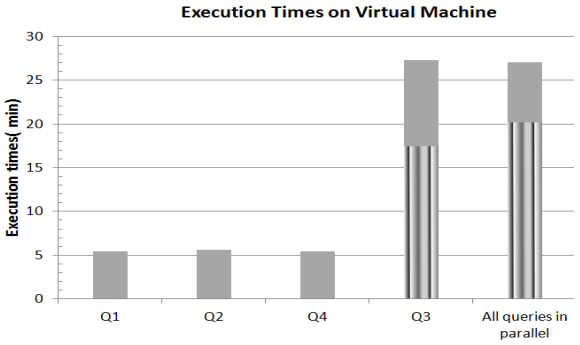

Figure 5. Performance

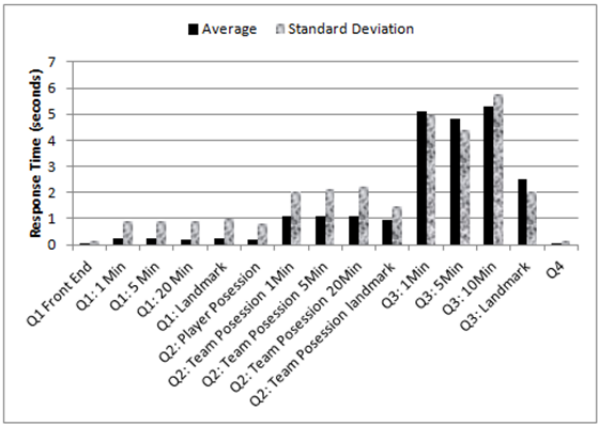

Figure 6. Delays

REFERENCES

[1] Bai, Y., Thakkar, H., Wang, H., Luo, C., and Zaniolo, C.: A Data Stream Language and System Designed for Power and Extensibility. *Proc. CIKM Conf.*, 2006.

[2] Botan, I., Derakhshan, R., Dindar, N., Haas, L., Miller, R. J. and Tatbul, N. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proc. VLDB Conf.*, 2010.

[3] Botan, I., Fischer, P. M., Florescu, D., Kossmann, D., Kraska, T., and Tamosevicius, R. Extending XQuery with Window Functions. *Proc. VLDB Conf.,* 2007.

[4] http://esper.codehaus.org/

[5] Law, Y-N, Wang, H., and Zaniolo, C.: Relational Languages and Data Models for Continuous Queries on Sequences and Data Streams. ACM TODS 36, 2, (May 2011).

[6] Li, J., Maier, D., Tufte, K., Papadimos,V., and Tucker, P. A. Semantics and evaluation techniques for window aggregates in data streams. *Proc. SIGMOD Conf.*, pp. 311 - 322, 2005.

[7] Patroumpas, K. and Sellis, T. Window specification over data streams. *Proc. EDBT Conf.,* 2006.

[8] Thakkar, H., Mozafari, B. and Zaniolo, C.: Designing an Inductive Data Stream Management System: the Stream Mill Experience. *Proc. 2nd International Workshop on Scalable Stream Processing Systems*, 2008.

[9] Zeitler, E. and Risch, T.: Massive scale-out of expensive continuous queries, *Proc. of the VLDB Endowment*, ISSN 2150-8097, Vol. 4, No. 11, pp.1181-1188, 2011