Grand Challenge: SPRINT Stream Processing Engine as a Solution*

Yingjun Wu[†1], David Maier[†‡2], Kian-Lee Tan[†3]
[†]School of Computing, National University of Singapore, Singapore 117417
[‡]Computer Science Department, Portland State University, Portland, OR 92701, USA
{yingjun[1], tankl[3]}@comp.nus.edu.sg, maier@cs.pdx.edu[2]

ABSTRACT

A stream processing engine, named *SPRINT*, is designed and implemented to efficiently process queries over high-speed sensor data streams from soccer games. SPRINT adopts several novel strategies, including a lock-free ring buffer, frame-based sliding windows, and dynamic parallel computation, to pursue three objectives: high speed, high precision, and low space consumption. Experiments show that SPRINT can achieve these three goals simultaneously.

Categories and Subject Descriptors

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

General Terms

Performance, Experimentation

Keywords

Stream Processing Engine, Complex Events, DEBS Grand Challenge

1. INTRODUCTION

The DEBS 2013 Grand Challenge provides an opportunity to develop a stream processing engine for high-speed sensor data analysis in real time. The challenge aims at processing multiple queries over high-speed sensor data streams from a soccer game. Solutions are evaluated from different perspectives including correctness, throughput, and innovation. To tackle this challenge, we designed and implemented *SPRINT*, a stream processing engine, to evaluate multiple continuous queries in parallel. Experiments show that SPRINT can elegantly handle complex queries while meeting the strict requirements of the challenge.

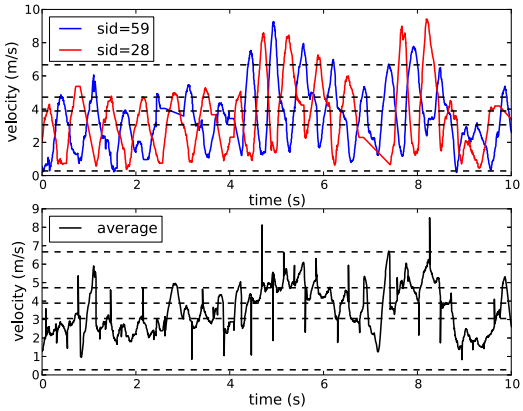

Figure 1: Sensor-generated velocity streams.

The paper is organized as follows. Section 2 generally reviews the problems and requirements proposed in the challenge and discusses the solution ideas. Section 3 describes the design and implementation of SPRINT in detail. Section 4 analyzes the performance of SPRINT from different perspectives. We discuss related work and conclude the paper in Section 5 and 6, respectively.

2. PROCESSING CHALLENGE QUERIES

The grand challenge requires us to evaluate four complex queries online. In this section, we generally describe the key ideas for solving these queries.

2.1 Query 1: Running Analysis

This query aims calculating instantaneous and aggregated running statistics for each player. The reported results need to satisfy two requirements: (1) all required statistics must be returned as streams at the frequency of at most 50 Hz; (2) a running-status interval with duration less than one second should be counted on top of the next intensity.

Sensor-generated instant velocity cannot directly reflect players' real running speeds. As stated in the problem description, each sensor generates streaming data of one leg's movement. Although the player's speed can be calculated as the average of two sensors, the obtained data are in fact dependent on the player's stride frequency, making it an

*This work is funded by the NExT Search Centre (grant R-252-300-001-490), which is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office.

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

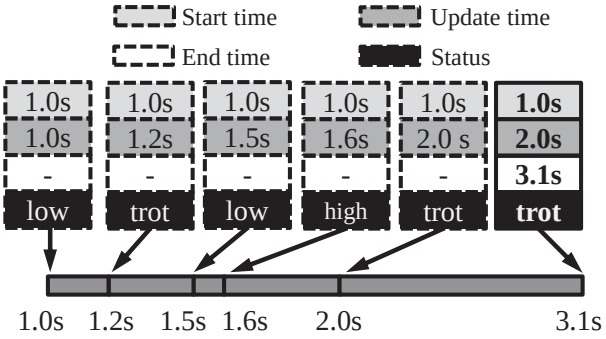

Figure 2: An example of the frame data structure.

unreliable indicator of running speed. Figure 1 illustrates this problem with two velocity streams generated from sensors with $sid = 59$ and $sid = 28$, both associated with the same player. The upper-half is the original data, and the lower-half is the corresponding average velocity stream of the two sensors. The series of dashed lines from bottom up represent the thresholds for each running status, i.e., *standing*, *trot*, *low*, *medium*, *high* and *sprint*. Obviously, the average velocity stream is not stable enough for reliable measurement. SPRINT uses two methods to address this problem. First, every running status with duration less than 0.1 second will be removed as noise. Second, additional cross-status sections are inserted between every two sibling status pair, i.e., *standing-to-trot*, *trot-to-low*, *low-to-medium*, *medium-to-high*, *high-to-sprint*, in which range the velocities can be regarded as either status. Consequently, data dithering can be controlled in a tolerable range, making the velocity stream stable.

To generate a reliable measurement, we introduce a special data structure, called a *frame* [5], with four fields attached: *start_time*, *update_time*, *end_time*, and *status*. Figure 2 illustrates the usage of frames. At first, as a new tuple comes in, we create a frame with *start_time* and *update_time* containing its timestamp, and *status* containing its running intensity. The frame remains unchanged until a tuple with new running intensity is detected. At this time point, if the difference between the current timestamp and *update_time* is less than 1 second, then the *update_time* and *status* fields are modified to the current timestamp and running intensity. Otherwise, the *end_time* is filled in with the current timestamp, marking the frame as a reliable measurement, and the frame is further reported and inserted into a compressed linked list with the *update_time* field omitted. As an example, the frame in Figure 2 will finally report a 2.1-second-long trot running status. Obviously, the frame data structure ensures any reported status is reliable.

In this query, every player has a corresponding linked list for the purpose of recording the running performance trends. Figure 3 gives an example of such a compressed linked list. To aggregate window-range statistics, a pointer is used to trace through the linked list and accumulate the results. This kind of linked list is also referred as a frame-based sliding window, which will be further elaborated in Section 3.

2.2 Query 2: Ball Possession

This query aims at calculating ball-possession percentages for each player as well as for the whole team. The key point

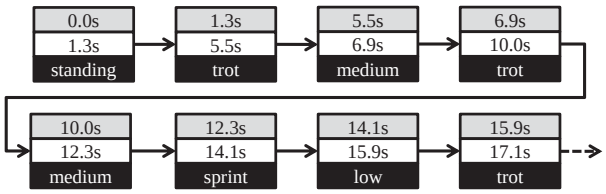

Figure 3: An example of compressed linked list for each player.

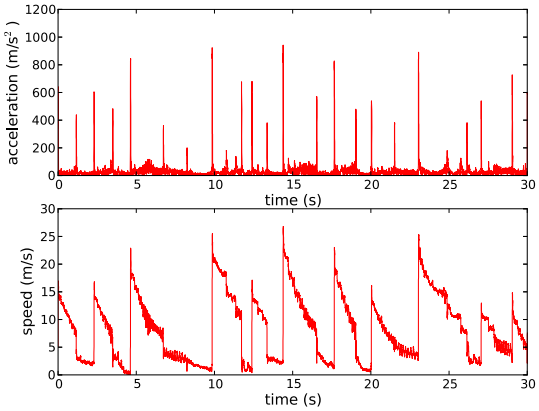

Figure 4: Ball acceleration and velocity. The upper figure shows the acceleration of the ball, and the lower figure shows the velocity of the ball at the same points in time.

is to efficiently detect the ball-hit event, which occurs when the ball acceleration peaks. Figure 4 shows the change of ball acceleration and velocity in a certain time interval. As suggested in the figure, the ball velocity oscillates with a regular pattern (possibly due to sensor rotation). Once the ball acceleration peaks, the velocity value changes drastically. As the changing pattern of ball velocity is more apparent and robust compared to the ball acceleration, SPRINT implements a velocity-based detection method to monitor the ball-hit events.

The velocity-based detection of ball-hit events continuously keeps track of the ball velocity. Once a sudden change in ball velocity is detected, SPRINT reports a ball-hit event, and then searches for the player who kicked the ball. Here, we define the "sudden change" as a change of +5 or -2 speed units (m/s) in 0.015 second. The player search uses an approximate nearest-neighbor search method. If no player is found in a one-meter-radius range, then the ball-hit event is simply treated as an outlier and discarded. Moreover, we use the "blind eye" method to improve processing efficiency. When the ball is kicked, the acceleration and velocity will stay at a high value for a certain time interval before dropping to normal. Therefore, once the ball-hit event is confirmed, SPRINT pauses velocity monitoring for 0.5 seconds and resumes it afterwards. All tuples in this half-second interval are ignored, since they are irrelevant to the event processing.

Similar to Query 1, the window-range statistics are also held in a frame-based sliding window. However, instead of maintaining linked lists for both teams, SPRINT only holds

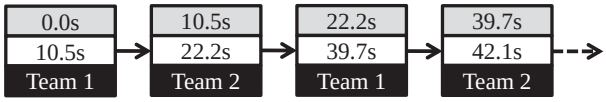

Figure 5: An example of compressed linked list for each team.

a single list to keep track of the possession-exchange event. The reason is that the possession lists for two teams can be serialized, as only one team can hold the possession at any time. An example of a frame-based sliding window in this query is illustrated in Figure 5. Obviously, the two teams control the ball alternately during the game.

2.3 Query 3: Heat Map

This query requires calculating statistics for a heatmap monitoring players' position distributions over a group of equally-sized cells partitioned from the entire field. Output streams should be generated with different window lengths for different combinations of parameters: 8×13 (a grid of 104 cells), 16×25 (800 cells), 32×50 (1,600 cells), and 64×100 (6,400 cells). A simple solution to this query is to partition the field into a grid of 6,400 (64×100) cells and store the corresponding data for each player to record how much time they spend in each cell, since the 6,400 cells are of the minimum granularity and can easily constitute the larger granularity cells. In other words, for each player, an array of 6,400 slots is maintained to record time spent within certain time intervals at each cell. Obviously, the output streams for the paramater combinations of 64×100, 32×50, and 16×25, can be accumulated by summing up the values from their corresponding smaller-granularity cells.

Obtaining heatmap values for the 8×13 combination is much more difficult, as its corresponding 104 cells cannot be exactly aligned with the 6,400 lower-granularity cells. To solve this problem, we store additional cells for the unaligned part and monitor the time span each player spends in each of these unaligned cells. When generating the results, data in corresponding cells are accumulated.

One problem in this query is that keeping track of all 6,400 cells for each player for a long time interval (10 minutes for example) consumes too much memory resource. Interestingly, this problem can also be solved with frame-based sliding windows, as described already for the previous two queries. A linked list is still maintained for each of the players. Within each frame, three fields are maintained: the cell ID, the start time the player enters the cell, and the end time the player leaves the cell. As most players only move in a certain small range and are unlikely to run across the whole field, the linked list for a particular player can be extremely short, greatly reducing memory consumption.

2.4 Query 4: Shot on Goal

The aim of this query is to detect when a player shoots the ball in an attempt to score a goal. Apparently, the set of goal-attempt events is a subset of the set of ball-hit events. We therefore first detect the ball-hit event with a separate detector, using the similar method presented in Query 2. As a result, the detection for a goal attempt will be delayed until a ball hit is determined. Once a ball hit is confirmed, we subsequently estimate whether the ball could reach the baseline within 1.5 seconds from its current position. If possible, then further computation estimates the ball's position at the exact time that it reaches the baseline, according to its current velocity and position. If the estimated position falls into the goal area, the corresponding tuple will be output until another ball hit is detected or the ball leaves the field. The procedure of detecting goal attempts is shown in Figure 6.

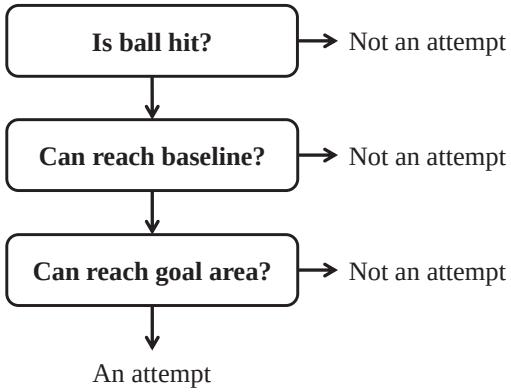

Figure 6: The procedure of detecting goal attempts.

3. ARCHITECTURE

This section describes the architecture of SPRINT, which is implemented in C++. To meet the strict requirements from the grand challenge, SPRINT largely takes advantage of multi-core techniques for the purpose of processing multiple queries in parallel. To optimize system performance, SPRINT adopts several innovative ideas, including a lock-free ring buffer, frame-based sliding windows, and dynamic computation method.

As shown in Figure 7, the overall architecture of SPRINT comprises three main components: preprocessor, shared ring buffer, and a group of parallel query processors. The preprocessor continuously reads tuples from the input data stream and feeds the parsed tuples into the shared ring buffer. The shared ring buffer is used to bridge and synchronize the incoming data and the query processing. Four independent parallel query processors run simultaneously for individual query tasks.

3.1 Lock-free Ring Buffer

SPRINT adopts the one-producer-multiple-consumer model to handle the incoming stream data. Traditionally, the producer-consumer model calls for expensive locking strategies to prevent data race. Instead, SPRINT follows the lock-free ring buffer model proposed in LMAX Disruptor [1]. The LMAX-style ring buffer brings two benefits to our SPRINT system. First, the circular buffer reuses allocated memory efficiently and thus helps to avoid memory allocation operations, which could lead to severe overhead in high-speed stream processing. Second, the lock-free ring buffer adopts CAS locks, introduced in C++ 11, to eliminate the expense incurred by traditional locks.

To maximize the memory bandwidth, we also set the message size to 4 KB, which is the default page size in most operating systems. The preprocessor (query processors) writes (reads) a 4 KB message block containing multiple tuples at one time, instead of writing (reading) tuples one by one. In

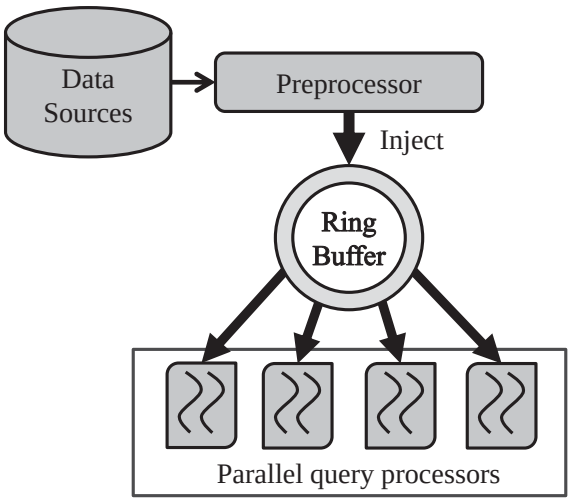

Figure 7: The architecture of SPRINT system.

this way, the time for memory fetching is minimized, further improving the system throughput.

3.2 Frame-based Sliding Windows

Generally, stream processing engines need to perform analysis in a sliding-window manner. One common method to implement sliding windows is to split the window into equal-sized subwindows, or named panes [4] [7], and gradually move the window foward. However, this approach can still be memory intensive. Consider Query 3. For each of the 16 players, 6,400 cells are maintained for every pane, with sliding interval equivalent to 1 second. Thus, to generate the aggregated statistics with window size of 10 minutes (600 seconds), $6400 \times 600 \times 16$ cells should be stored in memory, which may be unacceptable due to the memory constraints.

To tackle this problem, we employ a new data structure, called a *frame*, to greatly reduce memory requirements. Our key idea is to use a linked list of frames to compress the content of sliding windows by only tracking state transitions. Adjacent panes with equal states are merged to eliminate redundant information. In each frame, we only maintain three fields for *start_time*, *status*, and *end_time* to record the start time and end time of the current state.

To better demonstrate the usage of frame-based sliding windows, we recall how we use such sliding windows in Query 1. To record the players' running status, a linked list is stored for each player. At the beginning, the linked list is empty. Once a reliable measurement is made, we encapsulate the measurement in a frame, containing the start time, end time, and running status, and insert it into the linked list. The length of the linked list keeps growing until the earliest frames can be removed. For example, if the largest window size required is 10 minutes, and the current timestamp is 20m 10s, then all frames with end time less than 10m 10s can be dropped.

To strictly meet the required reporting frequency, we can also set a constraint on the time span of a certain frame, computed as $end\_time - start\_time$. For example, if we want to report the instant running intensity with a frequency of 50 Hz, then we can constantly check the equation $end\_time - start\_time \leq 0.02$ for every update of the temporal frame.

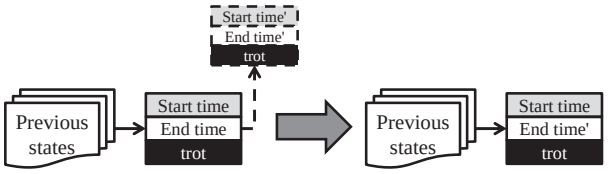

Figure 8: The merging procedure.

Once the constraint is violated, the frame is reported and an attempt made to insert it into linked list. If the *status* field is exactly the same as the end frame of the linked list, then the two frames will be merged. Figure 8 illustrates such a merging procedure.

To report the aggregated statistics for a certain window interval, we only need to search for the corresponding starting frame and calculate the statistics through the list. Note that for each query, we only maintain one frame-based sliding window with the size equivalent to the longest required window range. If three different window-range aggregations are needed, we then hold three pointers in the sliding window, indicating the appropriate starting frame for each.

Frame-based sliding windows also work with the other queries for the purpose of calculating window-range statistics. Note that other fields, such as the *update_time* field in Query 1, can be included in the frames if necessary.

3.3 Parallel Computation

Our system effectively employs multi-core techniques. We implement multi-core computation at two different levels: inter-query and intra-query. At the inter-query level, multi-core computation helps perform four independent queries concurrently. It mainly benefits from the lock-free ring buffer described above, as well as the dynamic computation discussed later. The intra-query level multi-core computation speeds up individual query processing by performing inner jobs in a parallel mode.

In SPRINT, Query 1 and Query 3 are easily parallelized. Figure 9 illustrates the parallel strategy adopted by SPRINT. As is shown in the figure, SPRINT follows a partition-and-merge paradigm to handle intra-query parallel computation. For Query 1 and Query 3, the input streams can be decomposed according to player ID, since the computation for each player is totally independent. To load-balance the parallel query threads, each processing thread is associated with an equal number of players. As the data comes in, each tuple is mapped to the corresponding processing thread by the job mapper. The processing thread then computes on-demand intermediate results. After that, the intermediate results are sent to a collector, where multiple results are combined and output streams are generated. Note that Query 3 can also be decomposed by cells. However, cell-based decomposition could lead to load-imbalance problems, so we do not incorporate it.

3.4 Dynamic Computation

SPRINT processes four queries in parallel using a global shared ring buffer. Each query uses its private pointer to fetch data from the shared ring buffer and then process it. A data slot is only feed after all four queries have read it. Consequently, as the processing of these four queries are of different speeds, the slowest one determines the consumption

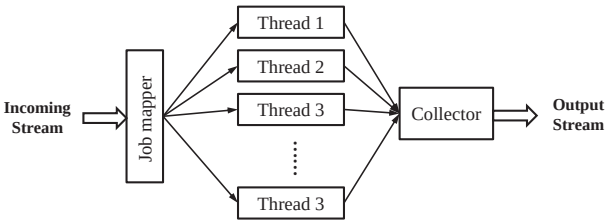

Figure 9: Intra-query level multi-core computation.

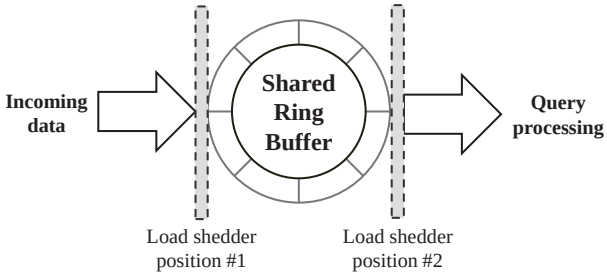

Figure 10: Load shedding strategies.

progress of the ring buffer. Given the high data rate, it is difficult to make the total throughput of four queries match the incoming data rate with accurate processing of every input data. Instead, a load shedding strategy is needed to adaptively match the consumption rate of the shared ring buffer with the incoming data rate, i.e., query processing should never block the incoming data due to filling of the ring buffer. Moreover, load shedding must take the workload balancing of the four queries into account, since different queries have different tolerance to load shedding.

The shared ring buffer is the central component for load shedding. In the architecture of SPRINT, the load shedder could be placed in two positions: either at the endpoint of the incoming data or the start point of the queries, as shown in Figure 10. Based on these possibilities, our load shedding strategy evolved as follows.

Global shedding. The load shedder is placed in position #1 in Figure 10 and controls load shedding by configuring a global shed factor. As long as the ring buffer is full, indicating congestion, SPRINT increases the global shed factor to drop a great percentage of the incoming data. The advantage of global shedding is its easy implementation. However, it is too crude, as it treats all four queries collectively as a single black box.

Speeding up the straggler. Instead of treating the queries as a black box, we open it up and check which query performs slowest, i.e., is the straggler [6]. The load shedder is placed in position #2 in Figure 10. Instead of reading data consecutively, each query skips its read pointer in the ring buffer to drop some data in order to bridge the relative processing speed gap with other queries. Each query owns a shed factor to control the skipping of its read pointer. When congestion is detected, we check which query is the straggler, and increase the corresponding shed factor. This operation continues iteratively until the congestion is eliminated. Compared with global shedding, this strategy is more precise, but still treats individual queries as black boxes.

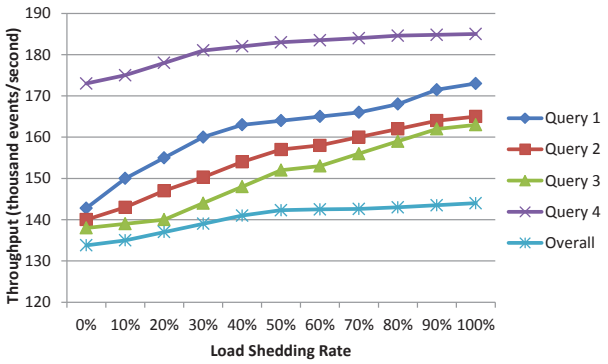

Figure 11: Throughput performance by counting input events.

Shedding based on tolerance. Different queries have different tolerances for load shedding. Thus we further open up the black boxes to control the load shedding more intelligently. Each query is configured with a *shed tolerance*, which is the maximum shed factor that the query can accept. We define the *tolerance distance* to be the difference between the shed tolerance and the current shed factor. When ring buffer congestion occurs, we increase the shed factor of a query with the largest tolerance distance at the time to alleviate the congestion. This operation is also done iteratively until the congestion is eliminated. Consequently, we are able to minimize the negative side-effects of load shedding. One remaining issue is that finding a suitable set of shed tolerances is nontrivial. In our implementation of SPRINT, we employ empirically determined values.

4. EVALUATION

In this section, we evaluate the SPRINT system from three perspectives: throughput, precision, and recall. All the experiments are conducted on a machine with four 2.00 GHz Intel processors and 2 GB memory. The underlying operating system is CentOS 5.8. We directly take the raw sensor dataset as the data stream source.

Throughput. In our experiments, *throughput* refers to the number of input events consumed per second. To measure the throughput of SPRINT, we first run each query separately to judge single-query performance. Then all the queries are run simultaneously to test overall system performance. The load-shedding rate is set manually and does not take advantage of the tuning strategy introduced in the previous section. To minimize the influence of ring-buffer size, we generate the results only after the throughput value stabilizes, i.e., when the ring buffer can be assumed full. Note that we have muted all the output information during the experiments, since the I/O performance can be quite slow and is usually system dependent.

Figure 11 shows the system throughput as the load-shedding rate varies. As is illustrated in the figure, the throughput of Query 4 is much higher than the other three queries, and remains relatively stable with the variation in load-shedding rate. The relation is Query 4 does not involve window-range aggregation operations, and its query logic tends to be quite simple. The overall throughput turns out to be close to the single-query throughput, as the performance of multiple queries benefits from the parallel computation strategy.

However, when the load shedding rate goes up, the overall throughput does not change too much, as parallel computation also brings synchronization overhead.

Precision and recall. As the ball possession and shot on goal statistics have already been provided, we can verify the correctness of Query 2 and Query 4 by comparing our results with the provided referee events. For Query 2, instead of evaluating the correctness of ball possession, we measure the ball-hit events. Table 1 presents the experimental results on precision and recall for Query 2 and Query 4. The load-shedding rate is variously set to 0%, 25%, 50%, 75%, and 100%. Obviously, no output is genereated when the load-shedding rate is set to 100%. As the load-shedding rate increases, the precision of the results remains high, and the recall also does not drop much even when the load-shedding rate reaches 75%. The major reason is that the input data rate is quite high (2000 Hz for the ball) and load-shedding does not prevent generating correct results.

Table 1: Experimental results on Queries 2 & 4.

	Load-shedding	Precision	Recall	F-Score
Query 2	0%	93.3%	90.8%	92.0%
	25%	93.3%	90.8%	92.0%
	50%	90.1%	87.9%	88.9%
	75%	91.5%	75.1%	82.5%
	100%	0.0%	0.0%	0.0%
Query 4	0%	84.2%	79.6%	82.0%
	25%	84.2%	79.6%	82.0%
	50%	83.3%	72.1%	77.3%
	75%	81.8%	62.8%	71.1%
	100%	N/A	0.0%	0.0%

As a reference, we also list the first 15 ball-hit events detected in Table 2. Note that, as our experiment shows, the referee events have around 3-5 seconds lag with the events in the dataset.

Table 2: First 15 ball-hit events.

Estimated Time	Reference Time	Player
0.01s	4.08s	Leo Langhans
2.31s	5.23s	Christopher Lee
4.64s	7.46s	Vale Reitstetter
9.86s	10.75s	Luca Ziegler
12.40s	15.74s	Vale Reitstetter
14.40s	17.47s	Christopher Lee
17.67s	20.48s	Kevin Baer
20.04s	23.07s	Christopher Lee
23.05s	25.53s	Luca Ziegler
29.04s	33.07s	Roman Hartleb
30.01s	34.16s	Erik Engelhardt
30.96s	35.00s	Roman Hartleb
47.47s	47.08s	Christopher Lee
50.21s	52.85s	Vale Reitstetter
53.22s	55.61s	Christopher Lee

To conclude, we have demonstrated the efficiency and effectiveness of SPRINT system in performing multiple complex queries in real time.

5. RELATED WORK

High-performace stream-processing systems are attractive in both academic and industrial communities. Representative works include Storm [2], a distributed real-time computation system, and StreamInsight [3], a commercial platform for complex event processing. These systems are general purpose but hard to optimize for the queries in DEBS 2013 Grand Challenge, since their underlying components are highly interdependent. Aiming at high performace, SPRINT is built from scratch and maximizes the efficiency of queries via several novel designs as mentioned in Section 3.

6. CONCLUSION

In this paper, we introduced a stream processing engine, named SPRINT, to tackle the DEBS 2013 Grand Challenge. SPRINT adopts several novel data structures, including a lock-free ring buffer and frame-based sliding windows, to evaluate the multiple online queries concurrently. The system can also dynamically tune the load shedding rate to adapt to the runtime environment. Experiments showed that SPRINT can simutaneously achieve the three requirements of high speed, high precision, and low space consumption. As future work, we want to further investigate missing-data problems, which occur quite frequently in sensor-generated data streams. We are also looking for a novel stream cleaning method to better filter out noise. Finally, we plan to upgrade our SPRINT system to a general distributed stream processing engine, which could enjoy better generality and scalability.

7. REFERENCES

[1] Lmax disruptor. https://github.com/lmax-exchange.

[2] Storm. https://github.com/nathanmarz.

[3] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola, X. Wang, D. Maier, S. Grell, O. Nano, and I. Santos. Microsoft cep server and online behavioral targeting. *Proc. VLDB Endow.*, 2(2):1558–1561, Aug. 2009.

[4] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *ACM SIGMOD Record*, 34(1):39–44, 2005.

[5] D. Maier, M. Grossniklaus, S. Moorthy, and K. Tufte. Capturing episodes: may the frame be with you. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 1–11. ACM, 2012.

[6] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 29–42, 2008.

[7] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.