A Performance Study of Event Processing Systems

Marcelo R.N. Mendes, Pedro Bizarro, Paulo Marques

CISUC, University of Coimbra,
Dep. Eng. Informática – Pólo II, 3030-290, Coimbra, Portugal
{mnunes, bizarro, pmarques}@dei.uc.pt

Abstract. Event processing engines are used in diverse mission-critical scenarios such as fraud detection, traffic monitoring, or intensive care units. However, these scenarios have very different operational requirements in terms of, e.g., types of events, queries/patterns complexity, throughput, latency and number of sources and sinks. What are the performance bottlenecks? Will performance degrade gracefully with increasing loads? In this paper we make a first attempt to answer these questions by running several micro-benchmarks on three different engines, while we vary query parameters like window size, window expiration type, predicate selectivity, and data values. We also perform some experiments to assess engines scalability with respect to number of queries and propose ways for evaluating their ability in adapting to changes in load conditions. Lastly, we show that similar queries have widely different performances on the same or different engines and that no engine dominates the other two in all scenarios.

Keywords: Benchmarking, Complex Event Processing, Micro-benchmarks.

1 Introduction

Complex Event Processing (CEP)[1] has emerged as a new paradigm to monitor and react to continuously arriving *events* in (soft-)real time. The wide applicability of event processing has drawn increased attention both from academia and industry, giving rise to many research projects [1, 2, 7, 16] and commercial products. CEP has been used for several purposes, including fraud detection, stock trading, supply-chain monitoring, network management, traffic monitoring or intensive care units control.

Most scenarios where event engines are being deployed are mission-critical situations with demanding performance requirements (e.g., high throughput and/or low latency). Interestingly, the range of scenarios is very broad and presents very different operational requirements in terms of throughput, response time, type of events, patterns, number of sources, number of sinks, scalability, and more. It is unclear what type of requirements demand more from engines, what happens when parameters are varied, or if performance degrades gracefully. To address the lack of event processing performance information, in this paper we make the following contributions:

[1] We use the terms “complex event processing”, "CEP" and "event processing" interchangeably. Likewise, we also use the terms "CEP system" and "CEP engine" interchangeably.

i. We present a number of micro-benchmarks to stress fundamental operations such as selection, projection, aggregation, join, pattern detection, and windowing (summarized in Section 3).
ii. We perform an extensive experimental evaluation of three different CEP products (two commercial, one open-source), with varying combinations of window type, size, and expiration mode, join and predicate selectivity, tuple width, incoming throughput, reaction to bursts and query sharing (Section 4).

2 Event Processing Overview

Like Data Stream Management Systems (DSMS) [1, 2, 16], CEP systems are designed to handle real time data that arrive constantly in the form of *event streams*. CEP queries are *continuous* in the sense that they are registered once and then run indefinitely, returning updated results as new events arrive. Due to low-latency requirements, CEP engines manipulate events in main memory rather than in secondary storage media. Since it is not possible to keep all events in memory, CEP engines use *moving windows* to keep only a subset (typically the most recent part) of the event streams in memory. In addition to these features shared with DSMS, CEP engines also provide the ability to define reactive rules that fire upon detection of specific patterns. Ideally, CEP engines should be able to *continuously adapt* their execution to cope with variations (e.g., in arrival rate or in data distributions) and should be able to scale by *sharing computation* among similar queries.

Section 2.1 lists the operations typically performed by CEP systems. We use these operations as the basis of the micro-benchmarks of Section 3.

2.1 CEP Characterization

A few event processing uses cases have been recently published [6], but it is still unclear which of them, if any, is representative of the field. There is, however, a core set of operations used in most scenarios and available, in one form or another, in all products:

- *Windowing;*
- *Transformation;*
- *Aggregation/Grouping;*
- *Merging (Union);*
- *Filtering (Selection/Projection);*
- *Sorting/Ranking;*
- *Correlation/Enrichment (Join);*
- *Pattern Detection.*

The performance of a CEP engine depends on: i) the algorithms implementing these basic operations; ii) parameters such as window type and size, and predicate selectivity; and iii) external parameters such as available resources, incoming data, and number and type of queries and rules.

2.2 Window Policies

Moving windows are fundamental structures in CEP engines, being used in many types of queries. Windows with different properties produce different results and have radically different performance behaviors. *Window policies* determine when events are inserted and removed (expired) from moving windows and when to output computations. Three aspects define a policy [12]:

i. **Window type**: determines how the window is defined. *Physical* or *time-based* windows are defined in terms of time intervals. *Logical*, *count-based*, or *tuple-based* are defined in terms of number of tuples [2].

ii. **Expiration mode**: determines how the window endpoints change and which tuples are expired from the window. In *sliding* windows endpoints move together and events continuously expire with new events or passing time (e.g., "last 30 seconds"). In *jumping* or *tumbling* windows the head endpoint moves continuously while the tail endpoint moves (jumps) only sporadically (e.g. "current month"). The infrequent jump of the tail endpoint of jumping windows is said to *close or reset the window*, expiring all tuples at once. In a *landmark* window one endpoint is moving, the other is fixed, and events do not expire (e.g., "since 01-01-2000").

iii. **Update interval (evaluation mode)**: determines when to output results: every time a new event arrives or expires, only when the window closes (i.e., reaches its maximum capacity/age), or periodically at selected intervals.

In general, commercial engines do not support all the combinations above.

3 Dataset and Micro-Benchmarks

In this section we describe the dataset used in our tests and summarize the micro-benchmarks in Table 2 (*a detailed description of the queries appears in Section 4*). We use a synthetic dataset because it allows exploring the parameter and performance space more freely than any single real dataset. The dataset schema is based on sample schemas available at the Stream Query Repository (SQR) [21]. In most application domains of SQR, event records consist in: i) an identifier for the entities in the domain (e.g., stock symbols in trading examples); ii) a set of domain-specific properties (e.g., "price", "speed", or "temperature"), typically represented as floating point numbers; and iii) the time when the event happened or was registered.

Based on these observations, we define the generic dataset schema of Table 1. The ID field identifies the entity being reported in the stream. The number of different entities, `MAX_ID` (ranges from 10 to 5.000.000), can greatly affect performance in joins, pattern matching queries, and aggregations. Tuple width is varied with the number of attributes A_i (from 1 to 125). The TS timestamp field is expressed in milliseconds and assigned by the load generator at runtime.

[2] There are also *semantic windows* whose contents depend on some property of the data (e.g., all events between events "*login*" and "*logout*"). We do not consider semantic windows in our study as none of the engines we tested implements them.

Table 1. Schema of the dataset used.

Field	Type	Domain
ID	int	Equiprobable numbers in the range (1, MAX_ID)
$A_1 ... A_N$	double	Random values following a uniform distribution U(1,100)
TS	long	Timestamp.

Table 2. Summary of micro-benchmarks.

Query	Factors under analysis	Metrics
Selection and Projection	• Selectivity: [1%, 5%, 25%, 50%] • # attributes: [5, 10, 25, 50, 125]	Throughput
Aggregations and Windows	• Window size (tuples): 500 to 500K • Window expiration: [*sliding*, *jumping*] • Aggregations: [*SUM*, *MAX*, *STDEV*]	Throughput
Joins	• Input Source: [stream, window, in-memory table, external table] • Input Size (# events): 500 to 100M • Join Selectivity: 0.01 to 10	Throughput
Pattern Detection	• Window Size (secs): 10 to 600 • MAX_ID: [100, 1k, 10k, 100k] • Predicate Selectivity: 0.1% to 10%	Throughput
Large Time-Based Windows	• Injection Rate (events/sec): 500 to100K • Window Size: 10 minutes to 12 hours	Throughput Memory consumption
Adaptability	(See Section 4.8.)	Maximum latency Latency degradation ratio Recovery Time Post-peak latency variation ratio
Multiple queries	• Number of Queries: [1, 4, 16, 64] • Window definition (size: 400k to 500k)	Throughput Memory consumption

4 Tests Specification and Results

In this section we discuss the results obtained after running the micro-benchmarks on three CEP engines. We emphasize that it is not our intention to provide an in-depth comparison of existing CEP engines, but rather to give a first insight into the performance of current products as a way to identify bottlenecks and opportunities for improvement. We focus on analyzing general behavior and performance trends of the engines (e.g. variations with respect to window size, tuple width, or selectivity).

4.1 Tests Setup

The tests were performed on a server with two Intel Xeon E5420 (*12M Cache, 2.50 GHz, 1333 MHz FSB*) Quad-Core processors (a total of 8 cores), 16 GB of RAM, and 4 SATA-300 disks, running Windows 2008 x64 Datacenter Edition, SP2.

We ran our queries on three CEP engines, two of which are developer's editions of commercial products and the other is the open-source Esper [11]. Due to licensing restrictions, we are not allowed to reveal the names of the commercial products, and will call engines henceforth as "X", "Y", and "Z". We tried multiple combinations of configuration parameters to tune each engine to its maximum performance (e.g., enabling buffering at client side, or using different event formats and SDK versions).

Figure 1 shows the components involved in the performance tests. Two slightly different architectures were employed. In either case, the load generation component communicates with an intermediary process called Adapter via plain socket, and CSV text messages. The Adapter then converts these messages into the native format of CEP engines and transmits them using their respective application programming interfaces (API). The difference between the two architectures shown in Figure 1 is that engines X and Z are standalone applications (*architecture 1*), while engine Y consists in a .jar file that is embedded into an existing application (*architecture 2*). This means that X and Z, receive/send events/results using inter-process communication, while Y uses lower-latency local method calls.

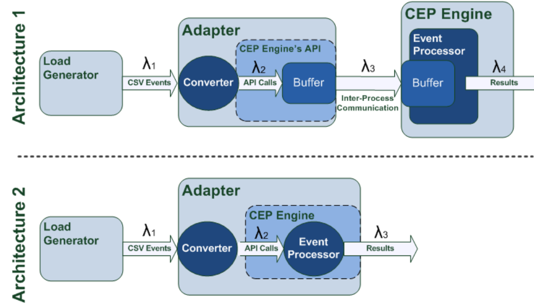

Figure 1. Architecture of evaluation setup

The input streams data were generated and submitted using the FINCoS framework [15], a set of benchmarking tools we have developed for assessing performance of CEP engines. Both the load generation components and the event processing engines under test ran in a single machine to eliminate network latencies and jitter. CPU's affinity was set to minimize interferences between the load generator, adapters and CEP engines. For all tests, unless otherwise stated, CEP engines ran in a single dedicated CPU core[3], while the load generator and Adapter ran in the remaining ones.

4.2 Methodology

Tests consisted in running a single continuous query at the CEP engine (*except for the multiple-query tests of Section 4.9*). They began with an initial 1 minute warm-up phase, during which the load injection rate increased linearly from 1 event per second to a pre-determined maximum throughput[4]. After warm-up, the tests proceeded for at least 10 minutes in steady state with the load generation and injection rate fixed at the maximum throughput. Tests requiring more time to achieve steady state (*e.g. using long time-based windows*) had a greater duration. All the measures reported represent averages of at least two performance runs after the system reaches a steady state.

[3] We verified that two of the engines did not automatically benefit from having more cores available. For the third engine, the version we tested was limited to use only one CPU core.

[4] The maximum injection rate was determined by running successive tests with increasing throughputs until CPU utilization was maximized or some other bottleneck was reached.

4.3 Test 1: Selection and Projection Filters

This micro-benchmark consists in two queries that filter rows (selection) or columns (projection) using query Q1 of Figure 2 (written in CQL [16]):

```
Q1: SELECT ID, A1,…,Am, TS
    FROM   stream1
    WHERE  ID <= K
```

Figure 2. Filtering query

The two data reduction queries vary the values of parameters `K`, `N`, and `m` (results in Figure 3). `K` is used to force desired selectivity, `N` is the number of input attributes and `m` is the number of projected output attributes (`m`≤`N`):

i. Row selection: varies predicate selectivity from 1% to 50%; `N=m=5`;
ii. Column projection: varies number of input attributes `N` from 5 to 125; `m` is fixed at 1 and row selectivity at 100%.

The throughputs achieved in this test series were very high, in millions of events per second. As expected, more selective predicates allow higher throughputs. The acute drop in performance in the projection query as the number of input attributes increases shows that tuple-width greatly affects performance. Notice that in both tests, Engine X was not fully utilizing the available resources (*utilization of its CPU was between 50% and 90%*) when its client API adapter became the bottleneck. Dedicating more CPU-cores to the adapter (*up to 7*) did not solve this issue.

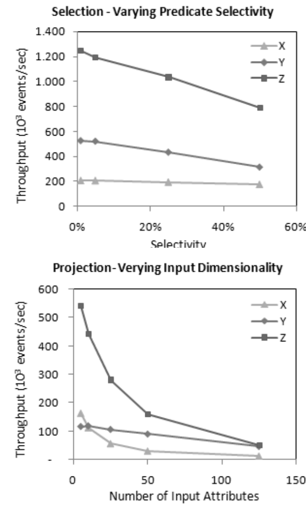

Figure 3. Filtering tests: Selection and Projection.

4.4 Test 2: Aggregation and Window Policy

The second micro-benchmark (query `Q2` in Figure 4) evaluates aggregations over different tuple-window configurations. (*Time-based windows are tested in sections 4.6 and 4.7.*)

```
Q2: SELECT ID, f(A1)
    FROM   stream1 [ROWS R Slide S]
    GROUP  BY ID
```

Figure 4. Aggregation Query, written in CQL [21].

We vary window size (parameter R from 500 to 500K), window type (parameter `S`=1 implies sliding window and parameter `S=R` implies jumping window), and aggregation function (parameter *f*=`MAX`, `AVG`, `STDDEV`, `MEDIAN`). Note that some functions can be computed at fixed cost (`STDDEV`, `AVG`) while others become more expensive as the window gets larger (`MAX` on sliding windows, or `MEDIAN`). Regard-

ing expiration mode, we expected *sliding* windows to be more expensive than *jumping* for two reasons. First, *sliding* windows expire tuples one-by-one while *jumping* windows expire them in batches. Second, *sliding* windows might need to keep more in-memory state (to deal with tuple-by-tuple expirations) while *jumping* windows may keep only counters and small summary data. Results are summarized in Figure 5.

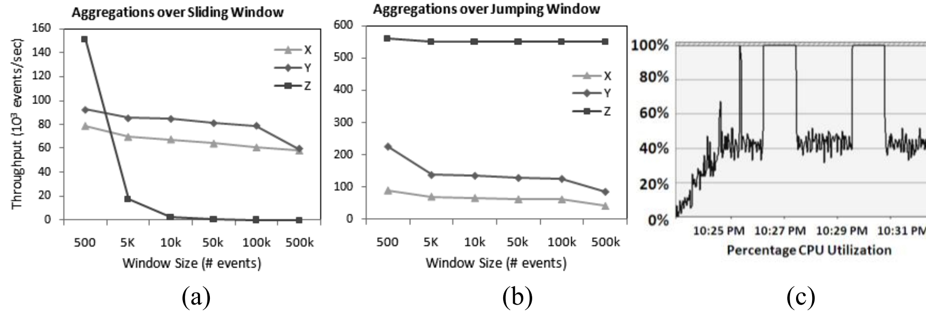

Figure 5. Aggregations Tests: varying windows sizes and policies on (a) sliding windows and on (b) jumping windows. Graph (c) is the CPU utilization of engine X for jumping windows.

Oddly, engine X had a worse performance with the *jumping* expiration mode than with *sliding* one. The cause seems to be inefficient batch-expiration of the *jumping* window tuples as shown by the peak CPU utilization coinciding with the periodic batch-expiration (Figure 5c). On engine Z, the performance difference between the two expiration modes was surprising: very high throughputs with *jumping* windows (the best of the three engines at around 550K tuples/second) but very low throughputs with *sliding* windows (the worse of the three, reaching only 50 tuples/second for windows of size 500K). For engine Y, results appears at first to meet our expectations, but in fact these two test cases are not directly comparable since Y's sliding windows output updated results for every tuple while its jumping windows update results only on window reset. Indeed, jumping windows showed a better performance not due to an implementation that benefit from the characteristics of this expiration mode, but rather, to a reduced evaluation/output frequency – *examining Y's open-source code we observed that the* `MAX` *aggregation is always computed by keeping the events of the window in a sorted structure; while this is a reasonable approach for sliding windows, it is inefficient for jumping windows, where* `MAX` *could be computed at constant cost*. Except for the aforementioned issue regarding computation of `MAX` on engine Y, varying the aggregation functions between `AVG`, `STDEV` and `MAX` generally had minor effects on performance of all engines. In contrast, all engines achieved considerably lower throughputs in the tests with the `MEDIAN` function. The `MEDIAN` function also showed to be more sensitive to window size than the other functions (e.g. see Figure 6).

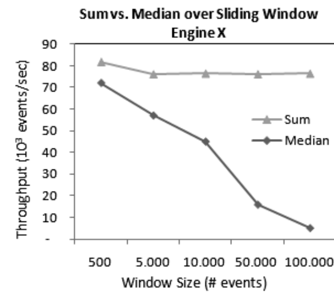

Figure 6. Median vs. Sum aggregates on engine X.

4.5 Test 3: Joins

This micro-benchmark evaluates join performance of CEP engines. We define three test series, each with different data sources and factors under analysis:

J1. Window-to-window join – joins two windows that are constantly being updated by event arrivals in the corresponding input streams;

J2. Stream-to-in-memory-table – simulates the situation where the content of an input stream must be enriched with static data stored as an in-memory table;

J3. Stream-to-DBMS-relation – table stored in an external database;

J1: Window-to-window

The window-to-window join query is the following:

```
Q3: SELECT * FROM  stream1 [ROWS S] AS S1,
                   stream2 [ROWS S] AS S2
    WHERE  S1.ID = S2.ID
```

Figure 7. Window-to-window Equi-Join Query

J1 series is comprised of three different tests as described below:

J1-1 *Varying window size and join selectivity*: parameter *S* varies from 500 to 500k. *MAX_ID* is held constant at 50k (i.e., the join is more selective for smaller window sizes);

J1-2 *Varying window size and keeping join selectivity*: parameters *S* and *MAX_ID* take the same values, from 500, to 500k, which ensures a fixed 100% join selectivity (each event finds a single match on the other window);

J1-3 *Varying join selectivity and keeping window size fixed*: *MAX_ID* takes the values 5k, 50k, 500k, and 5M while parameter S is held at 50k (each event finds, on average, 10, 1, 0.1 and 0.01 matching events on the other window).

Figure 8 below shows the results for this test series.

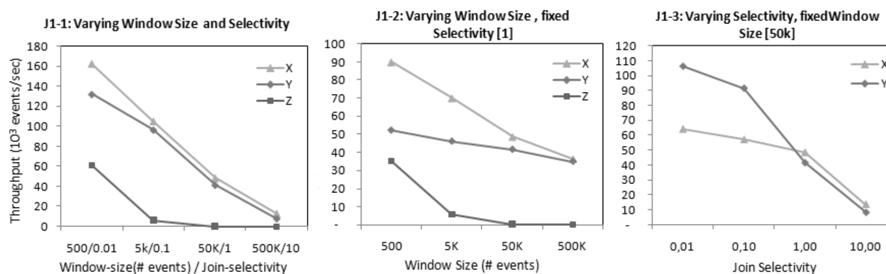

Figure 8. Tests Join: Window-to-window.

In J1-1, the acute drop in throughput was expected (due to increases in join input and selectivity) although the performance of engine Z degraded much faster than the performance of other engines. (*Recall that* engine *Z showed performance issues on pre-*

vious tests with aggregations over sliding windows, which seems to indicate that it has some problems with this expiration mode.)

Tests J1-2 and J1-3 reveal that engine X is more sensitive to window size while engine Y performs very well when join selectivity is low, but degrades more quickly when it gets close to or exceeds 1. For engine Z, we ran a modified version of J1-3, with a smaller window (size 500, not shown) in order to minimize the cost of window maintenance but there were no noticeable performance differences when varying the join selectivity, indicating again that sliding windows are not efficiently handled by Z.

J2/J3: Stream-to-in-memory-table and Stream-to-DB-relation

The queries of tests J2 and J3 have the following format:

```
Q4: SELECT * FROM   stream1 AS S,
                    table1  AS T
    WHERE  S.ID = T.ID
```

Figure 9. Stream-to-table Join Query

Figure 10 shows the corresponding results. In both tests a stream “S” with 4 fields is joined with a static table with 10 fields. In J2 the CEP engine is responsible for maintaining the table in main memory and for performing the join. In J3 the table is stored in an external database, which becomes responsible for the join (*every new event in stream S fires a parameterized query to the DBMS*[5]). The number of records in the table ranged from 1k to 10M (in-memory) and from 1k to 100M (DB); join selectivity is always 100% (*every event in the stream is matched against one and only one record in the table*).

In series J2, engine Y could not complete the test with 10M because it ran out of memory (prolonged garbage collections made it unresponsive). It is also interesting to notice how Z had a better join performance when operating over a table rather than over sliding windows (*see J1-2, in Figure 8*). In J3, two facts are worth mentioning: first, neither the CEP engines nor the DBMS were in their processing limits; the bottleneck was primarily the communication between these two components. Second, the performance was virtually unaffected from 1k to 1M as the DBMS was able to buffer the entire table into main memory. From this point on, the presence of IO calls, significantly lowered the query throughput.

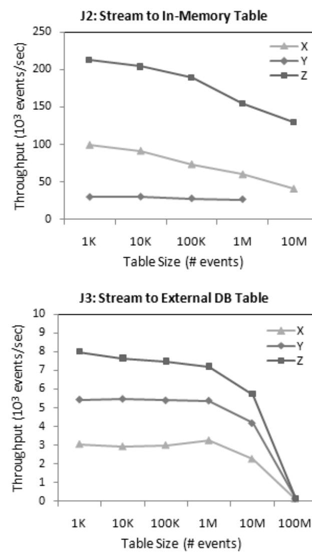

Figure 10. Tests Join between stream and table (J2) in-memory or (J3) in external database.

[5] We tested both with MS-SQL Server™ 2005 and Oracle™ 11g, and the results were similar.

4.6 Test 4: Pattern Matching and Negative Pattern Matching

We used query `Q5` and `Q5n` below to test pattern matching. `Q5` searches for instances of two events with the same ID in a time-based window of size *interval*, where the "A1" attribute of the second event is above some constant *K*. `Q5n` searches for sequences of an event not followed by a corresponding event within an interval.

```
Q5: PATTERN SEQ(A a1, A a2)      Q5n: PATTERN SEQ(A a1, ~(A a2))
    WHERE a1.id = a2.id AND            WHERE a1.id = a2.id AND
          a2.A1 > K                          a2.A1 > K
    WITHIN interval                    WITHIN interval
```

Figure 11. Sample Pattern Matching Queries (expressed using the SASE+ language [23])

The purpose of the "`a2.A1>K`" predicate is to verify that CEP engines indeed benefit of predicates in pattern detection by pushing them earlier in query plan construction. This micro-benchmark exercises three factors:

i. *Varying Window Size*: parameter *interval* ranges from 10 to 600 seconds. *MAX_ID* is held constant at 10k and *K* ensures a selectivity of 0.1%;

ii. *Varying Cardinality of Attribute ID*: *MAX_ID* ranges from 100 to 100k. *interval* was held constant at *1 minute* and *K* ensures selectivity of 0.1%;

iii. *Varying Predicate selectivity*: the predicate selectivity varied from 0.1% to 10%, while *interval* was held at *1 minute* and *MAX_ID* at 10k.

Figure 12 show the results of `Q5`. In the first experiment, all the engines had a very similar decrease in throughput as *interval* got larger. We could not determine the performance of engine Z for windows of sizes above 5 minutes because it consumed all available memory before tests could reach steady state (*the edition we tested was limited to address at most 1.5GB of memory*). As expected, increasing the cardinality of the correlation attribute ID *decreases* query cost, since less tuples pairs will have matching IDs. Similarly, more selective predicates (lower percentages) yield better performance as less tuples are considered as potential patterns matches.

`Q5n` showed to be less expensive than `Q5`. This difference has to do with the consumption mode [10] used in these tests (*all-to-all*): in the case of negative patterns a single occurrence of an event "a2" eliminates many potential matches. (*results of tests with negative pattern have been omitted due to space constraints*).

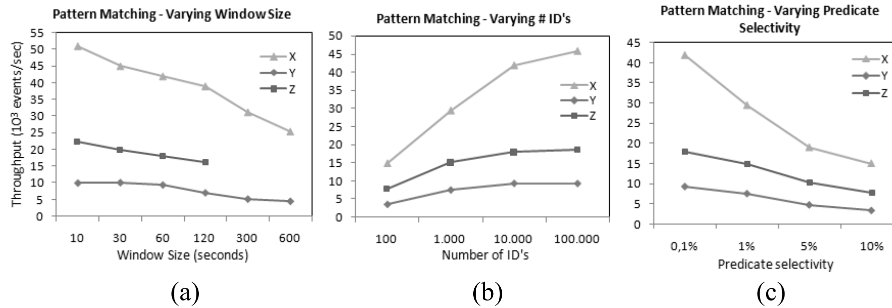

Figure 12. Pattern matching Tests varying (a) window size; (b) *#IDs*; (c) predicate selectivity.

4.7 Test 5: Large Time-Based Windows

Large time-based windows over high throughout sources may quickly drain system resources if all incoming events need to be retained. For example, one hour of 20-byte-size events on a 50k event/sec stream represents around 3.4 GB. Fortunately, certain applications require results to be updated only periodically, say every second, rather than for every new event (see Q6 bellow). In that case, for *distributive* or *algebraic* functions [14], Q6 can be rewritten in the equivalent query Q7.

```
Q6: SELECT AVG(A1)
    FROM   A [RANGE 1 HOUR]
    OUTPUT EVERY 1 SECOND;

Q7: SELECT SUM(s1)/SUM(c1)
    FROM (SELECT SUM(A1) AS s1, COUNT(A1) AS c1
          FROM A[RANGE 1 SECOND]
          OUTPUT EVERY 1 SECOND
          ) [RANGE 1 HOUR];
```

Figure 13. Two versions of aggregation query over time-based window with controlled output

Query Q7 computes 1-second aggregates on the inner query and 1-hour aggregates over the 1-second aggregates with the outer query. The space requirements of Q7 are:

*Inner window: (50000 events/second * 20bytes/event) * 1second = 977KB +*
*Outer window: (1 tuple/second * 20bytes/tuple) * 3600 seconds = 70KB*

This micro-benchmark runs Q6 and Q7 for large different window sizes and varying input rates. The goal is to verify if: i) Q6 is internally transformed into Q7; and ii) if not, to quantify the benefits of such transformation. The results of Q6 and Q7 for a 10-minute window appear in Table 3.

Table 3. Memory consumption (in MB) of CEP Engines for Q6 and Q7 (*10-minute window*)

	Input Rate			
Engine/Query	500	5,000	50,000	100,000
X, Q6	187	1,553	Out-of-memory	Out-of-memory
X, Q7	39	40	64	98
Y, Q6	455	3,173	Out-of-memory	Out-of-memory
Y, Q7	139	141	1,610	1,652
Z, Q6	56	64	56	55
Z, Q7	69	68	77	91

Observe that in engines X and Y query Q7 indeed reduced memory consumption when compared to Q6. In fact, Q6 showed a near-linear growth with respect to input rate and as such, engines X and Y exhausted memory (more than 13GB) for input rates above 50k events/sec even on small 10-minute windows. Engine Z had its memory consumption virtually unaffected by the input rate and almost identical in both

query versions; these results made us suspect at first that Z could be the only engine applying the query transformation automatically.

We then ran a second series of experiments with much larger windows. Input rate was kept at 100k events per second and window size was progressively increased up to 12 hours. The durations of these tests were always 1.5 times the window size. For engines X and Y, we ran the tests with `Q7`. For engine Z we tested both versions. Table 4 summarizes the results.

Table 4: CEP Engines' memory consumption for very large time-based windows (MB)

	Window Size				
Engine/Query	20 min	1 hr.	2 hrs.	6 hrs.	12 hrs.
X, `Q7`	114	128	141	146	147
Y, `Q7`	5,275	5,303	5,232	5,362	5,279
Z, `Q7`	70	73	55	58	52
Z, `Q6`	63	58	46	48	48

This new experiment exposed a behavior of engine Z not revealed in previous tests. While in the first experiments Z was able to keep memory consumption roughly unaffected by the number of events in the window, in this second series of tests, the CPU utilization and consequently maximum throughput were severely affected by the window size. As shown in Figure 14, `Q6` had a drastic drop in maximum throughput as window size was increased, while `Q7` showed a very steady throughput curve. It is worthy to point out that in the tests with `Q6` CPU was pushed to its maximum (*for windows of 20 min and beyond*), while with `Q7` CPU utilization stayed always around 1%. These numbers indicate that Z also does not perform the transformation mentioned above, but rather has an alternative implementation which sacrifices maximum throughput to keep memory consumption controlled.

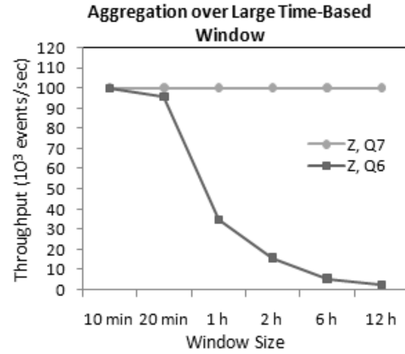

Figure 14 – `Q6` and `Q7` aggregations over large time-windows (engine Z).

4.8 Test 6: Adaptability to Bursts

The objective of this micro-benchmark is to verify how fast and efficiently the CEP engines adapt to changes in the load conditions. Although many factors may cause variations in the execution of continuous queries, here we focus solely on input rate. The tests of this series consist in:

i. An 1-minute warm-up phase during which the injection rate is progressively increased until a maximum value λ that makes CPU utilization around 75%;
ii. A 5-minute steady phase during which the injection rate is kept fixed at λ;

iii. A 10-second "peak" phase during which the injection rate is increased 50% (to 1.5λ), making the system temporarily overloaded;

iv. A 5-minute "recovery" phase in which the injection rate is again fixed at λ;

The query used is Q2, shown in Figure 4. To characterize the adaptability of CEP systems we define the following metrics:

- Maximum peak latency (Max_RT_{peak}): maximum latency either during or after the injection of the peak load;
- Peak latency degradation ratio ($RT_Degradation_{peak}$): 99.9th-percentile latency of peak phase with respect to 99.9th-percentile latency of steady phase:
  $$99.9th_RT_{peak}/99.9th_RT_{steady}$$
  In other words, what is the increase in latency caused by the peak?
- Recovery Time ($\Delta\tau_{recovery}$):
  $$\tau_{recovery}-\tau_{peak}$$
  where $\tau_{recovery}$ represents the timestamp of the first output event after peak injection whose latency is less than or equal the average latency of the steady phase and τ_{peak} is the timestamp of the last input event of the peak phase. That is, how long does it take for to return to the same latency levels?
- Post-peak latency variation ratio: Average latency after recovery divided by the average latency during steady phase:
  $$RT_{after_ecovery}/RT_{steady_phase}$$
  That is, what is the state of the system after it recovers from the peak?

Discussion: Blocking/Non-Blocking API and Latency Measurement

Recall from Figure 1 that events are sent to engines through API calls. On engine X, those API calls are non-blocking while on engines Y and Z they are blocking. In practice this means that X continues queuing incoming events even if overload while Y and Z prevent clients from submitting events at a higher rate than that they can process. As shown in Figure 15, there are multiple ways of computing latency. In order to properly measure latency for blocking calls, it is necessary to employ the "*creation time*" of input events instead of their "*send time*" – formula (3) in Figure 15. This formula allows accounting for the delays introduced by the blocking mechanism of the client APIs, which otherwise would pass unnoticed if we employed the moment immediately before sending the event.

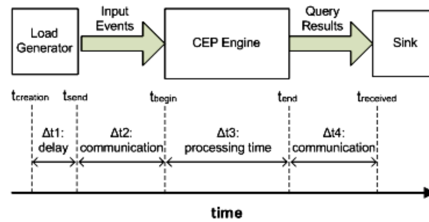

Figure 15: Latency Measurement.

Results

Table 5 and Figure 16 show the results of the adaptability test. Engine X, which adopts a non-blocking posture in the communication with clients, took much longer to recover from the peak and had a higher maximum latency than the two blocking engines, Y and Z. Nonetheless, after recovery, all engines returned to virtually the same latency level as that observed before the peak.

Table 5. Results for Adaptability Tests

	Engine		
Metric	X	Y	Z
Max_RT_{peak} (ms)	4.725,0	1.262,0	1.483,0
$RT_Degradation_{peak}$	82,8	57,4	5,9
$\Delta\tau_{recovery}$ (ms)	43.039,0	1.308,0	1.544,0
$RT_Variation_{post}$	1,0	0,9	1,0

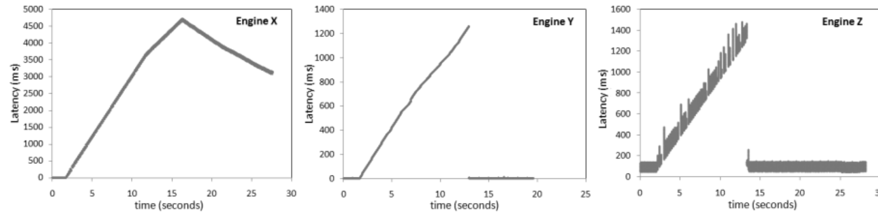

Figure 16. Adaptability Test: Scatter plot of latency before, during and after the peak.

4.9 Test 7: Multiple Queries (Plan Sharing)

The objective of this micro-benchmark is to analyze how the CEP engines scale with respect to the number of simultaneous similar queries. The query used in this experiment is a window-to-window join similar to Q3 (Figure 7). We tested two variations:

- **Test 1: Identical queries**. In this test we focus on computation sharing and the main metric is hence throughput. Window size is fixed in 1000 rows. To keep output rate fixed (*1 output per input event*), all queries have a predicate whose selectivity increases as we add more queries;

- **Test 2: Similar queries with different window sizes**. In this test we focus on memory sharing, so windows are large enough to observe differences when we increase the number of queries (in the range [400k-500k events]) and the injection rate is low so that CPU does not become a bottleneck;

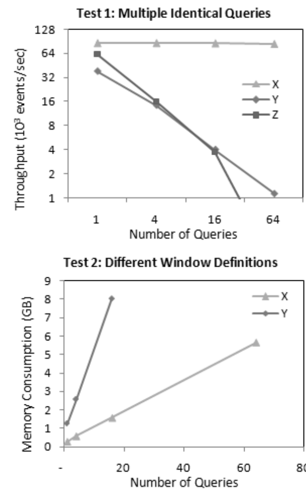

Figure 17: Multiple Queries Tests

The results of these two tests are shown in Figure 17. Engine X is the only one to implement some kind of query plan sharing: in the first test its throughput remained unaffected when the number of queries was increased. However, in the second test, in which queries were similar but different, it was not able share resources. Results also indicate that engines Y and Z do not implement query plan sharing. In fact, Y and Z could

not finish some tests of the second series: Y ran out of memory for 64 queries and Z become unresponsive while the window was being filled.

5 Related Work

Up to now, little previous work focused on the performance evaluation of event processing systems. White et al. [22] present a performance study which shows the latencies of a commercial CEP product while handling large volumes of events. Dekkers [8] carried out some tests for evaluating pattern detection performance in two open-source CEP engines. None of them characterize how query options such as window size and policy, or selectivity affects performance, nor covered more than one or two query type(s) or CEP product(s).

Some benchmarks have been proposed in areas related to CEP such as the BEAST benchmark [5] for active databases, or the *Linear Road* [3] or NEXMark [17] benchmarks for data stream management systems. However, these benchmarks measure only steady state performance for a fixed number of queries, and do not consider issues such as adaptability and query plan sharing. *SPECjms2007* [18] is a benchmark produced and maintained by the Standard Performance Evaluation Corporation (SPEC) aimed at evaluating the performance and scalability of JMS-based messaging middlewares. *SPECjms2007* thus focus on the communication side of event-driven systems rather than on query processing, which distinguishes it from our work.

6 Conclusions and Future Work

In this paper we presented a performance study of event processing systems. We proposed a series of queries to exercise factors such as window size and policy, selectivity, and event dimensionality and then carried out experimental evaluations on three CEP engines. The tests confirmed that very high throughputs can be achieved by CEP engines when performing simple operations such as filtering. In these cases the communication channel – *in our tests, the client API* – tends to be the bottleneck. We also observed that window expiration mode had a significant impact on the cost of queries. In fact, for one of the tested engines the difference in performance between jumping and sliding windows in one test was about 4 orders of magnitude. With respect to joins, tests revealed that accessing data stored in databases can significantly lower the throughput of a system. Pre-loading static data into CEP engine offers good performance and may thus solve this issue, but this approach is feasible only when data do not change often and fit in main memory. The tested engines had disparate adaptability characteristics. We observed that the approach used to receive events from clients – *either blocking or non-blocking* – plays a fundamental role on that aspect, although further investigation is still required to fully understand this topic (*e.g., testing bursts of variable amplitudes and durations or having changes in other parameters such as data distributions*). Finally, the tests with multiple queries showed that plan sharing happened only in one CEP engine and only for identical queries (*we still plan to broaden the investigation of this topic by incorporating tests with other*

classes of queries). It was also quite surprising and disappointing to realize that CEP engines were not able to automatically benefit from the multi-core hardware used in our tests. In general terms, we concluded that no CEP engine showed to be superior in all test scenarios, and that there is still room for performance improvements.

References

1. Abadi, D. J., et al.: Aurora. A New Model and Architecture for Data Stream Management. VLDB Journal, Vol. 12, August 2003, 120-139.
2. Arasu, A., et al.: STREAM: The Stanford Stream Data Manager. In Proc. SIGMOD 2003.
3. Arasu, A., et al. Linear Road: A Stream Data Management Benchmark. In Proc. of VLDB 2004.
4. Babcock, B., et al. Models and Issues in Data Stream Systems. In Proc. of SIGMOD 2002.
5. Berndtsson M., et al. Performance Evaluation of Object-Oriented Active Database Management Systems Using the BEAST Benchmark. In Theory and Practice of Object Systems, v.4 n.3, p.135-149, 1998
6. Bizarro, P., et al. Event Processing Use Cases. Tutorial, DEBS 2009, Nashville USA.
7. Chandrasekaran, S., et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In Proc. of CIDR 2003.
8. Dekkers, P. Master Thesis Computer Science. Complex Event Processing. Radboud University Nijmegen, Thesis number 574, October 2007.
9. DSAL Real-Time Event Processing Benchmark, http://www.datastreamanalysis.com/images/Real-Time%20EP%20Benchmark.pdf
10. Chakravarthy, S, Mishra, D. Snoop: An Expressive Event Specification Language for Active Databases. Data Knowl. Eng. (DKE) 14(1):1-26 (1994)
11. Esper, http://esper.codehaus.org/
12. Golab, L., Özsu, M. T.: Issues in data stream management. SIGMOD Record 32(2): 5-14 (2003).
13. Gray, J. (editor). The Benchmark Handbook for Database and Transaction Processing Systems, 2nd Edition. Morgan Kaufmann, 1993.
14. Gray, J, et al. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. Data Min. Knowl. Discov. 1(1): 29-53 (1997).
15. Mendes, M.R.N., Bizarro, P., Marques, P. A Framework for Performance Evaluation of Complex Event Processing Systems. In Proc. of DEBS 2008.
16. Motwani, R., et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In Proc. of CIDR 2003.
17. NEXMark Benchmark. http://datalab.cs.pdx.edu/niagara/NEXMark/
18. Sachs K., et. Al.: Workload Characterization of the SPECjms2007 Benchmark. In Proceedings of EPEW07, volume 4748, pages 228--244. Springer, 2007.
19. STAC-A1 Benchmark, http://www.stacresearch.com/council
20. STAC Report: Aleri Order Book Consolidation on Intel Tigertown and Solaris 10. Available at: http://www.stacresearch.com/node/3844
21. Stream Query Repository, http://infolab.stanford.edu/stream/sqr/
22. White, S., Alves, A., Rorke, D. WebLogic event server: a lightweight, modular application server for event processing. In Proc. of DEBS 2008.
23. Wu, E., Diao, Y., Rizvi, S. High Performance Complex Event Processing over Streams. In Proc. of SIGMOD 2006.