Marcelo Rodrigues Nunes Mendes

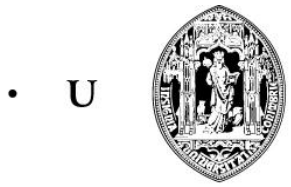
Performance Evaluation and Benchmarking of Event Processing Systems

Tese de Doutoramento em Ciências e Tecnologias da Informação, orientada pelos Professores Doutores Pedro Gustavo Santos Rodrigues Bizarro e Paulo Jorge Pimenta Marques, e apresentada ao Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Setembro de 2013

· U C ·

Universidade de Coimbra



UNIVERSIDADE DE COIMBRA

Performance Evaluation and Benchmarking of Event Processing Systems

Marcelo R. N. Mendes

under supervision of

Prof. Dr. Pedro Bizarro
University of Coimbra

Prof. Dr. Paulo Marques
University of Coimbra

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Informatics Engineering

September 2013

This research has been supported in part by the
Portuguese Science and Technology Foundation
(FCT), under grant Nº 45121/2008, and by a
European Commission FP6 Marie Curie
International Reintegration Grant.

– To my family –

Abstract

This thesis aims at studying, comparing, and improving the performance and scalability of event processing (EP) systems.

In the last 15 years, event processing systems have gained increased attention from academia and industry, having found application in a number of mission-critical scenarios and motivated the onset of several research projects and specialized startups. Nonetheless, there has been a general lack of information, evaluation methodologies and tools in what concerns the performance of EP platforms. Until recently, it was not clear which factors impact most their performance, if the systems would scale well and adapt to changes in load conditions or if they had any serious limitations. Moreover, the lack of standardized benchmarks hindered any objective comparison among the diverse platforms. In this thesis, we tackle these problems by acting in several fronts.

First, we developed FINCoS, a set of benchmarking tools for load generation and performance measurement of event processing systems. The framework has been designed to be independent on any particular workload or product so that it can be reused in multiple performance studies and benchmark kits. FINCoS has been made publicly available under the terms of the GNU General Public License and is also currently hosted at the Standard Performance Evaluation Corporation (SPEC) repository of peer-reviewed tools for quantitative system evaluation and analysis.

We then defined a set of microbenchmarks and used them to conduct an extensive performance study on three EP systems. This analysis helped identifying critical factors

affecting the performance of event processing platforms and exposed important limitations of the products, such as poor utilization of resources, trashing or failures in the presence of memory shortages, and no/incipient query plan sharing capabilities.

With these results in hands, we moved our focus to performance enhancement. To improve resource utilization, we proposed novel algorithms and evaluated alternative data organization schemes that not only reduce substantially memory consumption, but also are significantly more efficient at the microarchitectural level. Our experimental evaluation corroborated the efficacy of the proposed optimizations: together they provided a 6-fold reduction in memory usage and order-of-magnitude increase on query throughput. In addition, we addressed the problem of memory-constrained applications by introducing *SlideM*, an optimal buffer management algorithm that selectively offloads sliding windows state to disk when main memory becomes insufficient. We also developed a strategy based on *SlideM* to share computational resources when processing multiple aggregation queries over overlapping sliding windows. Our experimental results demonstrate that, contrary to common sense, storing windows data on disk can be appropriate even for applications with very high event arrival rates.

We concluded this thesis by proposing the *Pairs* benchmark. *Pairs* was designed to assess the ability of EP platforms in processing increasingly larger numbers of simultaneous queries and event arrival rates while providing quick answers. The benchmark workload exercises several common features that appear repeatedly in most event processing applications, including event filtering, aggregation, correlation and pattern detection. Furthermore, differently from previous proposals in related areas, *Pairs* allows evaluating important aspects of event processing systems such as adaptivity and query scalability.

In general, we expect that the findings and proposals presented in this thesis serve to broaden the understanding on the performance of event processing platforms and open avenues for additional improvements in the current generation of EP systems.

Resumo

Esta dissertação tem por objetivo estudar e comparar o desempenho dos sistemas de processamento de eventos, bem como propor novas técnicas que melhorem sua eficiência e escalabilidade.

Nos últimos anos os sistemas de processamento de eventos têm tido uma difusão bastante rápida, tanto no meio acadêmico, onde deram origem a vários projetos de investigação, como na indústria, onde fomentaram o aparecimento de dezenas de *startups* e fazem-se hoje presentes nos mais diversos domínios de aplicação. No entanto, tem-se observado uma falta generalizada de informação, metodologias de avaliação e ferramentas no que diz respeito ao desempenho das plataformas de processamento de eventos. Até recentemente, não era conhecido ao certo que fatores afetam mais o seu desempenho, se os sistemas seriam capazes de escalar e adaptar-se às mudanças frequentes nas condições de carga, ou se teriam alguma limitação específica. Além disso, a falta de *benchmarks* padronizados impedia que se estabelecesse qualquer comparação objetiva entre os diversos produtos. Este trabalho visa preencher estas lacunas, e para isso foram abordados quatro tópicos principais.

Primeiramente, desenvolvemos o *framework* FINCoS, um conjunto de ferramentas de *benchmarking* para a geração de carga e medição de desempenho de sistemas de processamento de eventos. O *framework* foi especificamente concebido de modo a ser independente dos produtos testados e da carga de trabalho utilizada, permitindo, assim, a sua reutilização em diversos estudos de desempenho e *benchmarks*.

Em seguida, definimos uma série de *microbenchmarks* e conduzimos um estudo alargado de desempenho envolvendo três sistemas distintos. Essa análise não só permitiu identificar alguns fatores críticos para o desempenho das plataformas de processamento de eventos, como também expôs limitações importantes dos produtos, tais como má utilização de recursos e falhas devido à falta de memória.

A partir dos resultados obtidos, passamos a nos dedicar à investigação de melhorias de desempenho. A fim de aprimorar a utilização de recursos, propusemos novos algoritmos e avaliamos esquemas de organização de dados alternativos que não só reduziram substancialmente o consumo de memória, como também se mostraram significativamente mais eficientes ao nível da microarquitetura. Para dirimir o problema de falta de memória, propusemos *SlideM*, um algoritmo de paginação que seletivamente envia partes do estado de *queries* contínuas para disco quando a memória física se torna-se insuficiente. Desenvolvemos também uma estratégia baseada no algoritmo *SlideM* para partilhar recursos computacionais durante o processamento de *queries* simultâneas.

Concluímos esta dissertação propondo o *benchmark Pairs*. O *benchmark* visa avaliar a capacidade das plataformas de processamento de eventos em responder rapidamente a números progressivamente maiores de *queries* e taxas de entrada de dados cada vez mais altas. Para isso, a carga de trabalho do *benchmark* foi cuidadosamente concebida de modo a exercitar as operações encontradas com maior frequência em aplicações reais de processamento de eventos, tais como agregação, correlação e detecção de padrões. O *benchmark Pairs* também se diferencia de propostas anteriores em áreas relacionadas por permitir avaliar outros aspectos fundamentais, como adaptabilidade e escalabilidade com relação ao número de *queries*.

De uma forma geral, esperamos que os resultados e propostas apresentados neste trabalho venham a contribuir para ampliar o entendimento acerca do desempenho das plataformas de processamento de eventos, e sirvam como estímulo para novos projetos de investigação que levem a melhorias adicionais à geração atual de sistemas.

Acknowledgments

I must thank a number of people who, either directly or indirectly, contributed for the conclusion of this work. First of all, I would like to thank my advisors Pedro Bizarro and Paulo Marques for all their guidance and for giving me the chance to live perhaps the period of greatest personal growth of my life. I also thank my former colleague Carlos Eduardo Pires for letting me know of and encouraging me to apply to the BiCEP research project, and professors Paulo Maciel and Ana Carolina Salgado, from Federal University of Pernambuco, for supporting my application. I have to thank also Roberto Torres for the chance of working on a very interesting project, which would later serve as inspiration during the development of the *Pairs* benchmark.

I also would like to thank my good friends Sven Stork, Ivano Irrera and Luis Pureza, who have accompanied me during a great part of my PhD. I also thank the countless amazing people from the most diverse corners of the planet that I had the luck to meet in Coimbra, which not only brought me a lot of joy and filled me with good memories, but also enriched my view of the world.

Last but not least, I would like to thank my family for everything they provided me through all these years. I thank my father Edson, for his incentive and support in going abroad to pursue my PhD. I also thank my brothers, Luciano and Luís, for inciting and feeding my will for excellence since my early days. I am also deeply thankful to my aunt Lucinha, for believing in our potential while still kids and supporting us whenever we needed. Finally I would like to thank my mother Fernanda for her love, care, and guidance, without which nothing that I have ever achieved would have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For more than thirty years, database systems have been the cornerstone of enterprise data management. However, the ubiquitous use of computing devices, the automation of once manual processes, and the popularization of Internet in recent years have brought data generation to unprecedented levels. Recent estimates from IBM indicate that 2.5 quintillion (i.e., 10^{18}) bytes are produced every day – so much that 90% of the data in the world today has been created in the last two years [30]. Following this outstanding information growth, more and more decision makers, in the most diverse domains, start to recognize the importance of continuously monitoring their businesses and infrastructures and respond immediately as the world changes. As a result, an entire class of novel applications has emerged, demanding automated and timely answers to new data as it arrives. It soon became evident that the classical database approach of persisting data first before it can be queried and emitting results only when explicitly asked by users was incompatible with this emerging paradigm.

The limitations of conventional data management platforms in dealing with those data-intensive, time-sensitive, applications led to the development of the *event processing engines*, a novel class of system specifically designed to meet users need for more agile data processing and analysis. Event processing (EP) systems provide the ability to extract valuable information from real-time continuous data sources, such as sensor

readings or stock market ticks, and promptly react to them. Contrary to regular databases, EP systems continuously produce updated query results as new data (events) arrive. The operations performed by these *continuous queries* can range from simple moving averages to the detection of complex patterns of events. EP systems also allow specifying *reactive rules* to determine which action must be taken upon the detection of a situation of interest. Typical reactions include updating dashboards, generating alerts (e.g., attempt of intrusion) or performing some task (e.g., sell a stock).

The concept of event processing exists for many years, but it was only recently that it has become a discipline by its own. Most of the research work in the area started in the mid to end of the nineties, under two independent fronts. The first formal attempt to make sense from the multitude of events happening at large-scale information systems was carried out at the RAPIDE [89] research project, from Stanford University. The goal of the project was to develop a language and a set of tools that allowed identifying timing and causal relationships among sets of seemingly unrelated events. At the same time, the database community started to realize that in many application scenarios data assumed the form of time-ordered *streams* rather than static datasets, with new pieces of information arriving continuously, usually at very high rates. Soon a whole new research area emerged, aimed at coping with the challenges posed by this new model of data processing. This resulted in the introduction of novel concepts and techniques, such as *continuous queries* and *sliding windows* [9], and the creation of a number of prototype *data stream management* systems (DSMS). From the University of Berkeley came the first general purpose DSMS: TelegraphCQ [23]. Shortly after, STREAM [11], the Stanford stream data manager, was released, and the Aurora project [1] was launched in a joint effort by Brandeis University, Brown University and MIT. Few years later many of those academic projects ended up turning into fully-functional products, some subsequently becoming major players in the event processing industry today (e.g., Progress Apama [74] and Streambase [94]).

The RAPIDE project did not evolve into a commercial product, but influenced considerably the field, with many of its proposed features being incorporated by event processing platforms of today (e.g., event pattern detection). As a consequence, the distinction between the two research fronts became less clear, and the systems started to be called by different denominations, including *Stream Processing Engines (SPE)*, *Event Stream Processing (ESP) systems*, *Complex Event Processing (CEP) system*s or simply *Event Processing systems*, depending on the context of the problem and the set of features supported by the platforms[1]. In spite of its different inheritances – and the divergences in nomenclature they might cause – , the last ten years have witnessed the consolidation of event processing as an important research discipline and industrial trend, with several specialized companies emerging and major technology providers, like IBM, Microsoft and Oracle, entering the market to offer their own solution.

1.1 Motivation and Problem Statement

As the technology matured, event processing platforms started to become increasingly prevalent in the most diverse domains of industry, including capital markets, telecom, healthcare, sensor networks, and many others [48]. It turns out that many of these event-driven applications are mission critical and, for most of them, the value of the responses provided by EP systems is proportional to their timeliness. For instance, in a variety of domains, such as *algorithmic trading* and *business activity monitoring*, identifying a trend or opportunity a few seconds or even milliseconds ahead of competition might mean the difference between success and complete failure. To make matters worse, EP systems are also expected to deal with massive amounts of data, usually coming from

[1] In this dissertation we use these terms interchangeably.

disperse event sources. In many application scenarios, input rates can be as high as thousands of events per second.

For this reason, it is fundamental to subject event processing platforms to rigorous performance analysis in order to guarantee that they are capable of meeting the stringent requirements posed by these event-driven applications. Nonetheless, there has been a general lack of information, evaluation methodologies and tools in which concerns the performance of event processing systems. Vendors have disclosed some performance numbers over the last years, but usually without the necessary details for replicating the results. Apart from that, only a few neutral studies have been published (e.g., [26], [27]), but they consisted in very simple tests and did not exercise the entire spectrum of features offered by event processing systems.

Furthermore, the event processing market today is very heterogeneous, with several competing products, each with their own functionality, query languages and implementation styles. It is therefore important to establish standard methods to compare them, so that users can be better informed when deciding which product best fits their needs. Traditionally, *benchmarks* have been used for this purpose, but for EP systems no proposal has been made up to date.

1.2 Research Objectives

This dissertation aims at addressing the aforementioned gaps by proposing standardized methodologies and tools that allow evaluating and comparing the performance of event processing platforms. In addition, this dissertation introduces a number of techniques to enhance the performance and scalability of EP systems.

1.3 Contributions

The main contributions of this dissertation can be grouped in four major areas:

- **Performance Evaluation Tools (Chapter 3)**

  A fundamental part of the performance evaluation process consists in developing tools for submitting load to the *system under test* (SUT) and gathering metrics from it. In the case of event processing systems, though, this is a particularly challenging task because of the significant differences found in the application scenarios and the heterogeneity of the available products. We address this issue by introducing FINCoS [63], a set of benchmark tools to assess the performance of the diverse event processing platforms under different test scenarios. In order to achieve that, the framework has been designed to be independent of any particular workload or product. Users can configure fully customizable synthetic workloads to stress specific aspects of event processing platforms or use real event traces to mimic their production environments. The framework can then be used to submit load to any product capable of exchanging events via the standard JMS API. This flexibility allows FINCoS to be used both in independent performance studies and also as a reusable component in multiple benchmark kits. Recently, the framework has undergone a thorough review process, having been accepted to integrate SPEC Research Group's repository of quantitative evaluation and analysis tools [84].

- **Performance Analysis (Chapter 4)**

  There has been very little information available about the performance of event processing systems, and the impact the different workload factors have on it. What are the bottlenecks? Will performance degrade gracefully in the presence of bursts? Will the systems scale appropriately as the number of simultaneous queries increases? Which product offers the best performance for a given workload scenario? In order to answer these questions, we propose a set of

microbenchmarks, and use them to conduct a thorough study examining the performance and scalability of three widely used EP engines. This work represented the first attempt in the area to systematize the evaluation of event processing platforms and is still to date one of the few disclosed studies where different systems were tested under comparable conditions.

- **Novel Algorithms and Optimization Techniques (Chapters 5 and 6)**

  The results of our performance analysis revealed important limitations in the current generation of event processing platforms. The problems ranged from poor utilization of computational resources to failures in the presence of memory shortages. We then propose novel algorithms and techniques to overcome those issues. In particular, our efforts concentrated in two areas: *i) better utilization of CPU and memory resources* (Chapter 5) by improving query execution at the micro-architectural level and employing more lightweight data structures and *ii) memory management* (Chapter 6), by introducing a paging algorithm that selectively offloads query state to disk when main memory becomes insufficient. We conduct extensive experimental evaluations to validate all the proposed techniques. In our experiments, the optimizations in CPU-RAM data path resulted in 6-fold reduction on memory consumption and order-of-magnitude increase on throughput for moving aggregation operations. Our experimental results also corroborated the efficacy of the proposed paging algorithm, which proved to sustain very high input rates (up to 300,000 events per second) for very large windows (about 30GB) while consuming small amounts of main memory (few kilobytes) and keeping latency under desirable levels (< 20ms).

- **Benchmarking (Chapter 7)**

  As noted in recent surveys [32] [39] [73], the event processing community has long resented the lack of standardized workloads that allow evaluating and

comparing the performance of the several platforms available. Some performance numbers have been made available by vendors (e.g., [33] [86] [103]), but each study employed its own workload, methodology and tools, which hinders any objective comparison among the products. In order to address this lack of standardized evaluation methods, we propose the *Pairs* benchmark. *Pairs* was designed to assess the ability of the EP systems in processing increasingly larger number of continuous queries and event arrival rates while providing quick answers – three quality attributes any event processing engine should possess. The final part of this thesis introduces the benchmark workload, metrics and tools. We also implement *Pairs* on two event processing engines and present a comparative performance study.

Chapter 2

Background

In this chapter we provide a broad overview on the topic of event processing systems. We start by describing their purpose, main characteristics and processing model. We then present the several implementation styles adopted by the different platforms of today and conclude by discussing the key performance aspects to consider when evaluating their performance.

2.1 Event Processing: An Overview

Essentially, event processing systems attempt to answer one question: "*What is going on in my business/infrastructure/information system right now*? The idea is to use the information contained in the thousands or millions of events happening on a given environment to gain insight about its current state and then react appropriately. For example, EP platforms have been widely used by analysts in capital markets to process the constant updates in stock prices in order to detect trading opportunities. By computing moving averages, correlating the current price with historical data, or simply following the sequence of price movements, they are able to determine if a certain stock is likely oversold or overbought and then, with that information, take the appropriate action (i.e., buy or sell the stock). Similarly, EP systems can be used in many other

scenarios to detect *undesired* situations, such as a network intrusion attempt or a critical health condition of patients in an intensive care unit.

To understand how event processing systems work, we need first to define accurately some terms. An *event* is defined as something that happens [62] or also as a significant change in the state of the universe [48]. From the perspective of an EP system, an event can be seen as an object that shall be subjected to computer processing [62]. This object consists in a record, with a number of attributes, containing information about the occurrence, much like a row in a relational database.

```
<Order>
   <attribute name="orderNo" value="12587"/>
   <attribute name="customerID" value="5341"/>
   <attribute name="itemID" value="28"/>
   <attribute name="date" value="12/04/2013"/>
   <attribute name="time" value="22:15:05"/>
<Order>
```

Figure 2.1: An "*Order*" event, represented as a XML, field-value pair, record.

Events can arrive from the outside world (e.g., sensor readings), be produced by information systems (e.g., ticks from electronic trading systems), or generated by the EP engine itself. In the first two cases, events are usually referred as being *simple* or *raw*, as they represent direct observations of the environment activity. In the latter, they are denominated *complex*, *composite*, or *derived* events, because they result from the composition of lower-level events. For instance, EP systems might respond to sequences of raw events such as SNMP traps and ATM transactions producing complex events like a network invasion alert or a fraudulent transaction warning. As illustrated in Figure 2.2, complex events are typically the ones that end-users are interested in[2].

[2] As a matter of fact, the discipline is often called "Complex Event Processing" precisely because one of its major goals is to identify such *complex events* from the myriad of seemingly unrelated simpler events.

Of course, not all patterns that users are looking for are a straight combination of low-level events. Very often EP systems are required to transform incoming events into high-level data first, by performing one or more intermediate operations, such as *aggregations* or *correlations*. For example, spotting a trading opportunity in capital markets might involve computing the average price of a stock over the last hour and then comparing this aggregate event with the stock price over the last week. In fact, these operations constitute a fundamental part of most event-driven applications and are at the core of the functionality provided by most EP platforms.

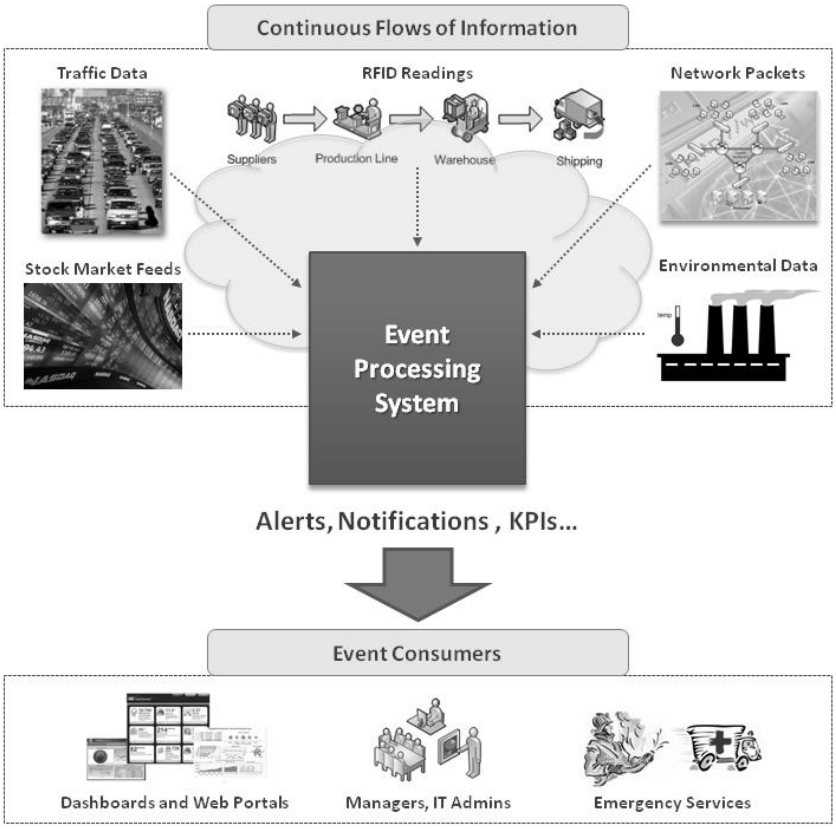


Figure 2.2: Event processing overview.

The main benefit of employing event processing systems lies in taking actions sooner, when they are more effective. For instance, the sooner a fraud occurrence is detected and countermeasures are taken, the lower the chances of significant financial loss.

Another advantage is that these actions are taken based on better-informed decisions, as EP systems are designed to rapidly extract and distill information from massive amounts of data that otherwise would be impossible for humans to analyze. Note that computer-aided processing of events has existed for many years, almost always implemented via special-purpose, custom-code software (e.g., proprietary algorithmic trading platforms and intrusion detection tools). EP systems, though, take the concept to a further level, by providing an abstraction layer that relieves developers from the burden of manually implementing efficient event processing logic. This results in shorter development cycles and usually better performance.

2.2 Event Processing Functionality

Event-driven applications come in the most diverse forms and shapes, and their functional and non-functional requirements tend to vary significantly from one domain to another. At the same time, EP systems differ considerably in their capabilities and implementation styles and for this reason it is frequently hard to delimitate precisely what consists the functionality of an event processing platform. Ultimately, event processing can be defined as any kind of computation that manipulates events. There is, however, a core set of event processing operations that are required by nearly all applications and are supported in a way or another by most products. Those include:

- ***Filtering***: the process of extracting information from event streams often starts by selecting which portions of the incoming records must proceed for further processing. This data reduction process can be carried out either *horizontally*, by discarding entire events and keeping only those that satisfy a given predicate, or *vertically*, by removing some attributes of each event. Note that these two operations are respectively equivalent to the relational operations *selection* (σ) and *projection* (π).

- ***Moving Aggregations and Windowing***: another common competence of event processing systems is to compute moving aggregations (i.e., AVG, COUNT, SUM, etc.) over event streams, automatically updating their results whenever new tuples arrive. The aggregation operation (Σ) is usually applied in conjunction with *moving windows* – data structures that retain only the most recently arrived events of a stream. Moving windows allow to limit the amounts of items (or time interval) to be considered when computing the aggregation function rather than using the entire set of events received since the beginning of execution (e.g., count the number of transaction records over the *last hour* or determine the average price of the *last three* updates for stock X).

- ***Correlation/Enrichment***: very often it is necessary to correlate events coming from different sources in order to obtain useful information. For instance, in order to detect non-ideal environmental conditions in a factory it might be necessary to merge readings from multiple sensor types (e.g., temperature and humidity). Another common scenario is to *join* ($\bowtie$) real time information carried by events with historical data stored in databases or data warehouses, with the purpose of *enriching* the incoming tuples or identifying deviations from the historically observed behavior.

- ***Event Pattern Matching***: one of the most fundamental features offered by event processing platforms consists in detecting sequences of events that together represent a situation of interest. An event pattern query is generally a statement that specifies *a set of constituent events*, their relative *order*, and a *time interval* within which the events must happen. For instance, an intrusion detection application might register a pattern query looking for a sequence of five consecutive failed login attempts, coming from the same remote terminal, within an interval of one minute. A slight variation of the concept, *negative patterns* look for the non-occurrence of events. For example, a fleet management software might need to emit a warning if a vehicle is known to have departed

but no corresponding notification informing of its arrival at the destination is received within its expected travel time.

- ***Integration***: event-driven applications need some form of integration with existing systems in order to receive events from external sources and data feeds, and output results to consumers. To address this need, most event processing platforms are bundled with a set of input and output adapters that allow them to communicate through diverse technologies and protocols (e.g., JMS, JDBC, FIX, RSS feeds, CSV files, etc.).

Note that although presented here separately, the aforementioned operations are typically strongly coupled in an event-driven application. In fact, most of such applications can be seen as a chain of those basic operations, through which events flow, as illustrated in Figure 2.3 below.

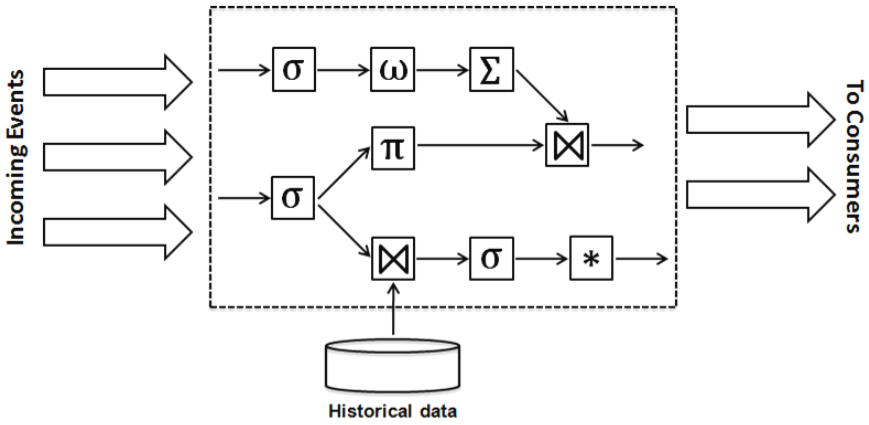


Figure 2.3: General structure of an event processing application.

Throughout the process, events are transformed, discarded, and new ones are created, until finally the answers of interest are produced, and delivered to the appropriate destinations.

2.3 Processing Model

EP systems adopt a processing model considerably different from that of conventional data management systems. These divergences are in fact a natural consequence of the different requirements posed by event-driven applications in comparison with regular transaction processing and analytical applications. For example, data is produced at much higher rates in event-driven applications (hundreds or thousands of events per second) than in classic OLTP and OLAP applications. The data also differs in nature. While databases deal with data that need to be stored for posterior access, the usefulness of events is usually limited to a short time after their occurrence. Therefore, contrary to conventional databases, which operate over persistent data stored on disks, event processing systems manipulate *transient data*, which is kept most of the time in *main memory* to ensure fast answers.

Another difference is how information is obtained from the data sources. Database systems adopt a *pull-based* approach, which requires applications to issue a query in order to retrieve data. EP systems, on the other hand, are designed to deal with applications that require automatic updated answers whenever new data arrives (*push-based* model). In that sense, the two approaches can be considered to be orthogonal: instead of storing data once and executing queries multiple times over it, in EP systems queries are *registered* once and then data is matched against them, producing a *continuous* flow of answers.

2.4 Implementation Styles

Due to their different inheritances, EP systems differ considerably in the way users express their event processing logic. Generally, vendors adopt one of three main design styles: *SQL-based query languages*, *composition rules* and *production rules* [31].

Most *SQL-based* event processing platforms have their roots in early academic research on data stream systems. They provide query languages similar to the SQL database standard, extended with some new elements to support streaming operations (e.g., windowing). Adopted by several products such as Esper [34], Oracle OEP [72], Streambase [94] and Sybase ESP (formerly Aleri) [97], the SQL-based approach is the prevalent implementation style today. The listing below, expressed with Esper EPL query language, illustrates the usage of this approach.

EXAMPLE: "*Retrieve the average temperature over the last minute for each room*":

```
select   roomId, avg(temperature)
from     SensorReading.win:time(1 minute)
group by roomId
```

Composition rules are used by pattern matching systems, like the commercial event processing platform RuleCore [79] and the academic prototype SASE [104]. With this approach, event patterns are specified by composing single events through predefined operators. These *composition operators* can be seen as functions whose input and output are streams of events. For instance, a *sequence* operator takes two streams A and B and produces a stream C of events whenever an event from A is followed by another from B. Other common composition operators are *conjunction*, *disjunction* and *negation*. The listing below exemplifies the specification of a pattern matching query using the SASE+ language:

EXAMPLE: "*Detect an uptrend in temperature*":

```
PATTERN SEQ (SensorReading s1, SensorReading s2, SensorReading s3)
WHERE       s1.roomId = s2.roomId AND s2.roomId = s3.roomId AND
            s1.temp > s2.temp AND s2.temp > s3.temp
WITHIN      1 minute
```

Production rules are at the core of Rete-based systems, like the Drools Fusion EP engine [29], and constitute an important part of the TIBCO Business Events platform [99]. A production rule is a statement consisting in two parts: a "WHEN" *condition* and a "THEN" *action*. Whenever the condition *becomes* true, the specified action is executed by the engine. The listing below shows an example of a production rule expressed using Drools Fusion language.

EXAMPLE: "*Emit alert in case average temperature rises above a given threshold*":

```
rule   "High-temperature alert"
  when
       TemperatureThreshold($max : max)
       Number(doubleValue > $max) from accumulate(
         SensorReading($temp : temperature) over window:time(1m),
         average($temp))
  then
       System.out.println("Room temperature above threshold!")
end
```

The heterogeneity found in the event processing landscape, where each of the many competing products adopt their own languages, architectures, data models, and processing techniques, reinforces the importance of standardized evaluation methods. An event processing benchmark can help identifying good and bad design decisions, which in turn might serve to improve existing systems and assist in the definition of standards. Recent efforts [20] [53] have achieved some advance on this topic by proposing methods to identify and conciliate the functional differences among the several products, but a standard event processing query language is still an open research issue.

2.5 Performance Concerns in EP Systems

The performance of event processing systems is generally measured in terms of *throughput* and *processing latency*. The former represents the number of events that an

EP system can process per unit of time, while the latter is the time it takes for the engine to produce a result after its triggering event has happened. These metrics have been commonly used because they allow measuring the ability of EP systems in meeting the two most critical performance requirements of event-driven applications, namely processing of massive amounts of data and timeliness[3].

Overall, these two main metrics are directly affected by the number and complexity of continuous queries running at the EP engine. In addition, their values tend to vary over time as a result of changes in the system and load conditions (e.g., state size of queries, garbage collection activity, selectivity of predicates). Thus, the performance of an EP system is closely related to its ability in processing increasingly larger numbers of concurrent queries and gracefully dealing with changes in load conditions. We discuss these two quality attributes next and briefly review how the problems have been addressed by previous work.

2.5.1 Query Scalability and Plan Sharing

In many scenarios, hundreds to thousands of continuous queries and rules might be running simultaneously at an EP engine, some of which may be very similar in terms of the computation performed or memory structures used. For instance, a stock trading system typically executes multiple strategies from diverse analysts, each monitoring a set of securities and with slightly different triggering conditions. Ideally, an event processing engine should be able to identify similarities between the configured queries

[3] It should be noted that different applications have different definitions for *timeliness*. The requirements differ not only on the length of the time span (i.e., minutes, seconds, milliseconds, etc.) but also on how consistently the answers are provided on time. For instance, in some scenarios it is satisfactory if the EP system is able to provide answers, on average, before a given threshold. Others, however, have more stringent requirements, demanding some guarantees that the specified deadline is not going to be missed. These different definitions affect the metric selection process, as in some cases it might be appropriate to use the average to summarize the different latency observations, but for others it might make more sense to use the maximum observed latency or a percentile measure.

and process them in a shared way. This allows the system to handle more gracefully increasing loads without having to resort to hardware upgrades.

As pointed out by early research in the data streams field [11], there are at least three components of a continuous query execution plan that can be shared:

i. *Operators*: queries that perform the same operation on the same incoming data can share the execution of the common operators, thus saving CPU cycles;
ii. *Intermediary queues*: intermediate results of internal operators can be shared among queries, thus saving memory space;
iii. *Synopsis structures*: likewise, shared synopsis structures (e.g., sliding windows) can help reducing memory space requirements.

Figure 2.4 illustrates the execution of two continuous queries, Q1 and Q2, in a shared way. The former is a selection over a join of two streams, R and S, while the latter is a join of three streams, R, S, and T. The join between streams R and S is common to both query plans, thus both the operator (O1) and its output queue (q3) can be shared among them.

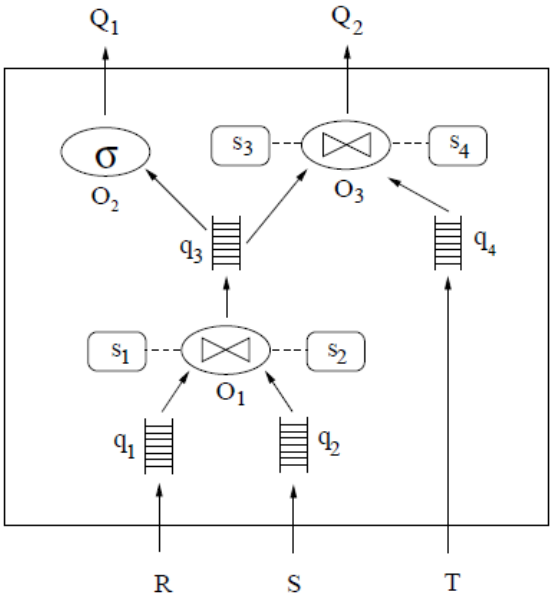


Figure 2.4: Sharing Continuous Query Plans [11].

Shared query processing has been subject of extensive research in the context of data stream systems [8] [11] [56]. A number of techniques have been proposed to efficiently share resources among stateful operations like sliding-window aggregates and joins. More recently, it has also been demonstrated that significant performance gains can be achieved by sharing both storage and computation when processing multiple pattern matching queries in event processing systems [3].

2.5.2 Adaptivity

Event processing applications run continuously for hours or even days without interruption, and as such it is very likely that the conditions change during their execution. For this reason, there has been considerable work on *adaptive* query processing techniques that allow stream processing engines to dynamically adjust their behavior in response to changes in load conditions. Essentially, the proposals aim at either improving the processing of continuous queries when their execution plan becomes sub-optimal [13] or dealing with overload conditions, for instance, by shedding load [1][57]. Recent work has also demonstrated the benefits of adaptivity in distributed configurations. Aniello et al [7] propose scheduling mechanisms for the Storm stream processing engine [92] that adapt their behavior according to the topology and runtime communication pattern of applications in order to reduce response time.

2.6 Summary

In this chapter we provided background information necessary for a clear understanding and better appreciation of this dissertation. The main concepts and characteristics of event processing systems were introduced. We described their functionality and processing model, and presented the main implementation styles adopted by the several products. We also discussed the key performance metrics of EP platforms, namely, throughput and processing latency, as well as closely related quality attributes such as query scalability and adaptivity.

Chapter 3

Benchmarking Tools for EP Systems

In this chapter we present the first of the four major contributions of this dissertation, namely addressing the lack of common tools for evaluating the performance of event processing platforms. For that, we propose FINCoS, a framework for load generation and performance measurement of EP systems. FINCoS leverages the development of novel benchmarks by allowing researchers to create synthetic workloads, and enables users of the technology to evaluate candidate solutions using their own real datasets. An extensible set of adapters allows the framework to communicate with different EP systems, and its architecture permits to distribute load generation across multiple nodes. FINCoS is used repeatedly in most experimental evaluations conducted throughout this dissertation. The framework is also publicly available for use by the general audience in [38] and [84].

3.1 Motivation and Contributions

As mentioned in section 1.1, event processing platforms are in many cases a central part of mission-critical applications, such as algorithmic trading, fraud detection, healthcare systems, and traffic control. In those scenarios, a failure to respond on time might cost lives or incur in severe material losses. It is therefore fundamental to subject EP systems

to rigorous evaluation in order to ensure they perform well and as expected even when faced with eventual fluctuations in load conditions.

However, evaluating the performance of event processing systems when standards, applications and capabilities of this evolving technology are not clearly defined is a challenging task. The diversity of application scenarios and the lack of standard benchmarks make it necessary to experiment with multiple test workloads. Moreover, the variety of products available, each adopting their own implementation style and query language, makes difficult to specify precisely the workload and the interfaces between the test infrastructure and the event processing platforms.

In that context, our goals with FINCoS were twofold: to reduce the amount of work necessary to carry out a performance evaluation study on EP systems and minimize the impact of the structural and functional divergences among the products on the process as a whole. The first goal was achieved by ensuring that synthetic workloads can be quickly devised and easily swapped from one test to another. Also, the framework provides common mechanisms for data generation, event scheduling, and performance measurement, freeing users from having to implement routines for that. This not only accelerates the process but also ensures that the diverse systems can be measured under comparable conditions, using identical methodologies and criteria. The second goal was achieved by means of an extensible set of *adapters*, which decouple most of the framework functionality from the event processing products.

The main contribution of FINCoS is to provide a unified approach through which diverse event processing systems can be evaluated and objectively compared independently of their inherent differences. This is beneficial for both users of the technology, which are now able to better assess the performance of their candidate platforms, and the academic community, which can more quickly develop and experiment novel benchmarks for the event processing field.

3.2 Architecture

The FINCoS framework is composed by five main components as shown in Figure 3.1.

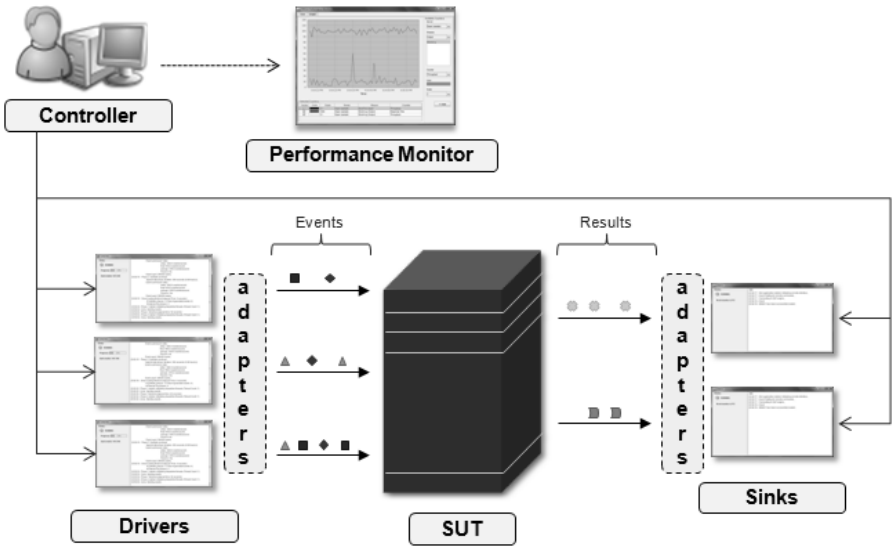


Figure 3.1: Overview of FINCoS components

Drivers simulate external event sources, submitting load to the system under test (SUT). On the opposite side, *Sinks* receive the results produced by the SUT, storing them in log files for subsequent answer validation and performance measurement. The communication with the EP engine in both cases is made through an extensible set of *adapters*, which convert the events produced by the framework into a data format understood by the target system and vice-versa. A graphical application, denominated *Controller*, allows users to configure, execute, and monitor performance tests. The results of these performance runs can then be visualized both in real-time and after test completion, using the *Performance Monitor* component.

3.3 Characteristics and Core Features

FINCoS provides a wide range of options in the definition of experimental evaluations. For instance, the execution of drivers can be split into phases, each with its own

workload characteristics (e.g., event submission rate, types and datasets). This is useful not only for breaking performance tests into well-defined parts (e.g., warm-up and measurement interval) but also for evaluating the ability of event processing platforms in adapting to changes in the load conditions. In addition, users can choose if events should be generated by the framework itself or read from files containing real-world event data. The former shall be useful for researchers studying the performance of event processing platforms while the latter should help customers trying to mimic their environments. The workload can also be seamlessly scaled by simply adding more *drivers* and *sinks* to the configuration.

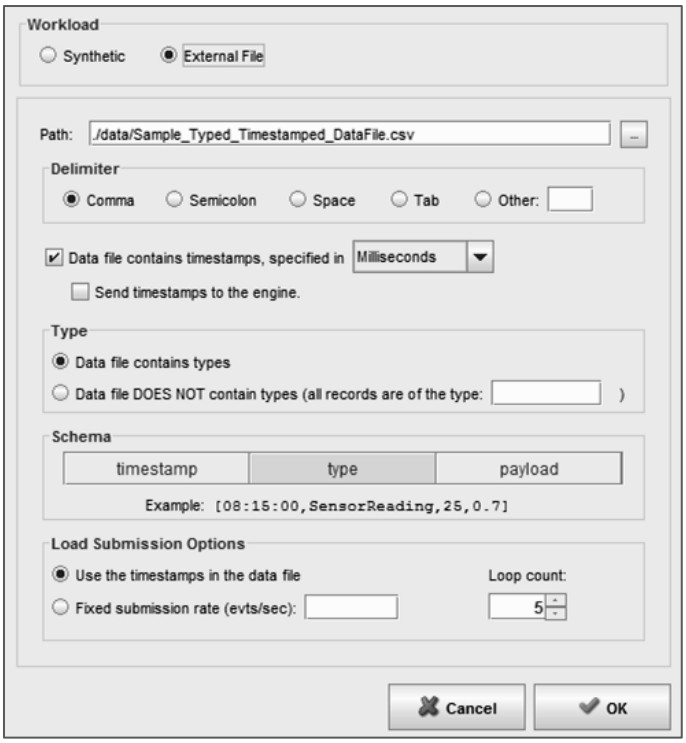


Figure 3.2: Configuration of a workload based on a user-provided data file.

Besides enabling users to define arbitrarily complex and realistic workloads, the framework was also designed to be portable across different EP products. FINCoS allows running performance tests with any EP system capable of exchanging events through a standard JMS middleware. In addition, the framework supports direct

communication with event processing platforms through custom adapters (using products APIs).

3.4 Performance Measurement

After test completion, the performance of the system under test is measured using the *performance monitor* application and the log files produced by *sinks* (the framework also allows measuring performance while tests are running at the cost of a slight overhead). The tool presents performance stats in both tabular and graphical formats – the former displays a snapshot of throughput and latency for each query running at the SUT, while the latter shows the evolution of these metrics over time. It is also possible to visualize the statistical distribution of the latency samples as shown in Figure 3.3.

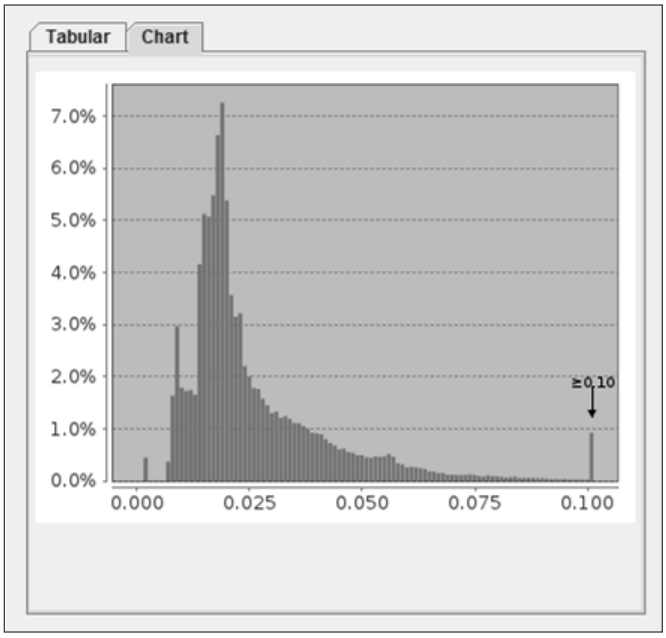


Figure 3.3: Latency histogram displayed by the FINCoS *Performance Monitor* tool.

Response time is measured by computing the difference between the time the SUT emitted a given result and the timestamp of the incoming event that triggered it. For that, output tuples produced by the CEP engine must explicitly include the timestamp of

the causer event (the timestamp of the result itself is automatically collected and appended by the framework).

Since there is a great variance in the way the several event processing platforms operate, FINCoS provides some flexibility for computing response time. Figure 3.4 illustrates the three possible definitions of response time supported by the framework.

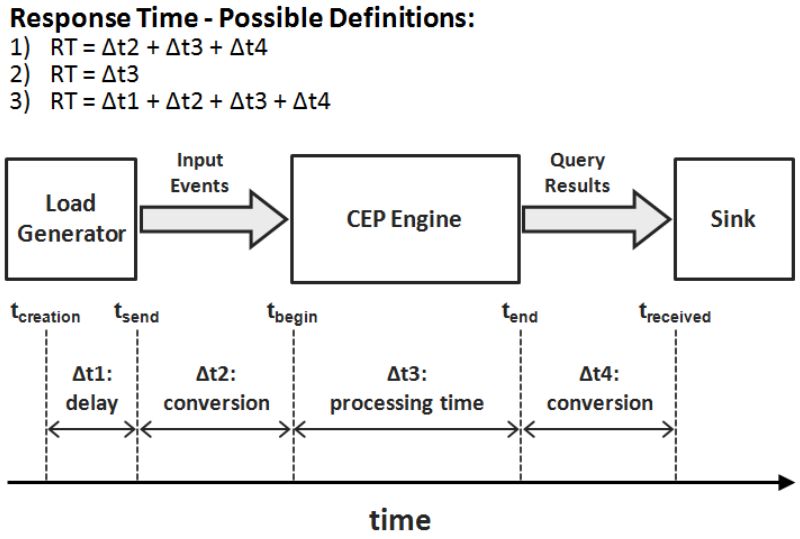


Figure 3.4: Latency measurement modes supported by FINCoS

The first, which we denominate *end-to-end* latency, represents the time it takes for an output tuple to arrive at a *sink* after the corresponding event that triggered it is sent by a *driver*. Note that in this definition the time for converting the event from the internal representation of the framework to a format understood by the SUT (and vice-versa) is accounted as part of the response time. Alternatively, the second definition can be used if the user wants to measure only the *processing time* of events inside the EP engine. In this case, the events are timestamped inside the *adapters*, immediately before and after sending and receiving events to the SUT. A third option is available and is intended for accounting for delays introduced when the dispatch of events blocks on their processing at the EP engine (the framework then uses the event scheduled time instead of a measured time).

In addition to the point where events are timestamped, FINCoS allows users to choose the resolution used to compute latency – either milliseconds or nanoseconds. Generally, the nanoseconds resolution should be preferred as it is more accurate and many EP systems offer sub-millisecond processing latencies. However, response times can be measured in nanoseconds only if *drivers* and *sinks* run in the same machine. It also incurs in more overhead than a millisecond resolution, so users should balance the need for accurate response time measurement and high event submission rates.

3.5 Summary

The FINCoS framework is a highly configurable tool that provides load generation and measuring capabilities for users and researchers who desire to carry out performance evaluations on event processing platforms. Fully-customizable workloads can be configured and tests can be performed on virtually any event processing platform, via standard JMS middlewares or directly, through customized adapters. These characteristics not only reduce substantially the amount of work involved in the evaluation of event processing systems but also leverage the development of novel benchmarks, as the framework can be reused as a portable component in multiple benchmark kits.

The first version of FINCoS was released in 2008 [63], and since then it has been considerably extended and improved. Earlier this year, the framework has undergone a rigorous review process, having been accepted to integrate SPEC Research Group's repository of peer-reviewed quantitative evaluation and analysis tools [84]. FINCoS is an open-source tool, and can be downloaded free of charge from the project web site [38]. A user guide and a tutorial with detailed instructions on how to use the framework are also available.

Chapter 4

A Performance Study of EP Systems

In the last chapter we proposed the FINCoS framework as a solution for the lack of standardized tools for performance evaluation of event processing platforms. In this chapter we focus on establishing a systematic evaluation methodology and analyze how several workload factors affect the performance of EP systems. For that, we propose a set of microbenchmarks to exercise the core aspects of EP platforms. The tests were designed to be simple and with clear queries semantics, so that they could be easily understood and replicated. We then execute the microbenchmarks on three different engines while we vary workload factors such as window size and policy, predicate selectivity, and injection rate. Among other things, results reveal that similar operations have widely different performances on the tested engines, and that improvements in some areas are required.

4.1 Introduction

There has generally been little information regarding the performance of event processing systems. Until recently, most of the available numbers had been provided by vendors, using tests designed by themselves (e.g., [33], [86], and [103]). Besides the obvious partiality issue, the lack of common workloads, metrics and methodologies and, in some cases, of details, hinder any objective comparison among the results obtained in

those studies and make it difficult to replicate them. Apart from those sponsored results, a neutral performance study of two EP platforms has been previously presented in [27]. The tests, however, were limited to event pattern matching, involved relatively low input rates, and were conducted in non-production hardware. Thus, it is still hard to say how EP engines will perform in more diversified or demanding situations, which factors affect most their performance, and where vendors should focus their optimization efforts. To make matters worse, the range of scenarios where event processing systems are being deployed 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 those parameters are varied, or if performance degrades gracefully when load conditions change.

4.1.1 Summary of Contributions

In this chapter we address this lack of information by conducting a comprehensive performance study where the diverse aspects of event processing platforms are exercised. In particular, we make the following contributions:

- We introduce a number of microbenchmarks to stress fundamental operations of EP systems, including selection, projection, aggregation, join, pattern detection, and windowing.
- We present the results of an extensive experimental evaluation of three widely-used EP 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.

The remainder of this chapter is organized as follows. Section 4.2 defines concepts and terminology necessary to understand the experiments. Section 4.3 introduces the

proposed microbenchmarks and section 4.4 describes the testing methodology. Results are presented in section 4.5. Finally, section 4.6 discusses related work and section 4.7 summarizes our conclusions.

4.2 Background

4.2.1 Window Policies

Moving windows are fundamental structures in EP 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 [40]:

- **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[4].

- **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 *tumbling* or *jumping* windows the head endpoint moves continuously while the tail endpoint moves (jumps) only sporadically (e.g. "current hour"). The infrequent jump of the tail endpoint of tumbling windows is said to *close or reset the window*, expiring all tuples at once. In a *landmark*

[4] 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, though, because two of the tested engines do not support them.

window one endpoint is moving, the other is fixed, and events do not expire (e.g., "since 8:00 AM").

- **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 practice, EP platforms do not support all the combinations above.

4.2.2 Event Pattern Matching

In pattern detection queries, an additional aspect to be considered is the *consumption mode* (also called *consumption policy*), which determines which event occurrences may be used for event composition when multiple candidates exist [22]. For example, consider the simple pattern matching query:

```
A -> B
```

The statement above looks for any situation when an event "A" is followed by an event "B". Assume that this query is registered in an EP engine and then the following event sequence is received:

$$A_1, A_2, B_1, A_3, B_2$$

(where A_1, A_2, and A_3 are three different occurrences of event A and B_1 and B_2 are two different occurrences of event B).

Research [22] describes two types of policy: *chronicle consumption policy* finds the first occurrence of each event necessary for the event composition and the *recent consumption policy* finds the most recent occurrence of each event necessary for the event composition.

However, in practice, the engines we tested do not offer exactly these consumption policies. Instead, they offer the possibility to mark the member events with keywords like `ALL` or `ONE`. The keyword `ONE` implies that an event can only be member of a

single composite event. The keyword `ALL` implies that events can be reused as members in multiple composite events. These two keywords can then produce four variations of the example composite event:

i. "`ALL A -> ALL B`"
Output: (A_1,B_1), (A_2,B_1), (A_1,B_2), (A_2,B_2), (A_3,B_2)

ii. "`ALL A -> ONE B`"
Output: (A_1,B_1), (A_1,B_2)

iii. "`ONE A -> ALL B`"
Output: (A_1,B_1), (A_2,B_1), (A_3,B_2)

iv. "`ONE A -> ONE B`"
Outputs: (A_1,B_1), (A_2,B_2)

As a matter of fact, consumption policies for pattern matching queries are one of the areas where event processing languages differ most. After examining the documentation of the engines tested in this study, we concluded that only the *all-to-all* policy had the exact same semantics across all of them.

4.3 Microbenchmarks

A few event processing uses cases have been published over the past years [17], but none of them is representative of the entire field. Nonetheless, as discussed in section 2.2, there is a core set of operations used in most scenarios, which are available, in one form or another, in all products, including:

- *Filtering (Selection/Projection)*
- *Windowing*
- *Aggregation*
- *Correlation/Enrichment (Join)*
- *Pattern Detection*

Thus, the overall performance of an event-driven application running at an EP engine shall depend on how efficiently those basic operations are implemented. In addition, workload parameters such as window type and size, and predicate selectivity will

determine how much work the EP engine must perform to answer the queries. Finally, external factors such as available resources, incoming data, and number and type of queries and rules might positively or negatively impact the system performance.

The purpose of the microbenchmarks introduced in this chapter is to evaluate the capacity of EP engines in processing those core event processing operations and quantify the effect that the different workload factors have on their performance. In addition, we also evaluate how well the engines adapt to changes in event arrival rates and scale with respect to the number of simultaneous queries. Table 4.2 summarizes the experiments conducted throughout this chapter (*a detailed description of each microbenchmark is provided in section 4.5*).

As input, 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) [93]. 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 shown in Table 4.1.

Table 4.1: 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.

The ID field identifies the entity being reported in the event stream. The number of different entities, `MAX_ID` (ranges from 10 to 5,000,000), can greatly affect the performance of joins, pattern matching queries, and grouped 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.

Table 4.2: Summary of microbenchmarks.

Query	Factors under analysis	Metrics†
Filtering	▪ Selectivity: [1%, 5%, 25%, 50%] ▪ # attributes: [5, 10, 25, 50, 125]	▪ Throughput
Aggregation and Windowing	▪ Window type: [*count-based*, *time-based*] ▪ Window size - count-based: 500 to 500K tuples - time-based: 10 minutes to 12 hours ▪ Window expiration: [*sliding*, *jumping*] ▪ Aggregation function: [`SUM`, `MAX`, `STDEV`] ▪ Injection Rate (events/sec): 500 to100K	▪ Throughput ▪ Memory consumption
Joins	▪ Input source: [*window*, *in-memory table*, *DB table*] ▪ Input size (# events): 500 to 100M ▪ Join selectivity: 0.01 to 10	▪ Throughput
Pattern Detection	▪ Window size (seconds): 10 to 600 ▪ Attribute cardinality: [100, 1k, 10k, 100k] ▪ Predicate selectivity: 0.1% to 10%	▪ Throughput
Adaptivity	▪ Injection rate	▪ Maximum latency ▪ Latency degradation ratio ▪ Recovery Time ▪ Post-peak latency variation ratio
Scalability	▪ Number of queries: [1, 4, 16, 64] ▪ Window size: 400k to 500k events	▪ Throughput ▪ Memory consumption

† *The definition and meaning of each metric is discussed later on this chapter.*

4.4 Setup and Methodology

4.4.1 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 EP engines, two of which are developer's editions of commercial products and the other is the open-source Esper [34]. 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 4.1 shows the components involved in the performance tests.

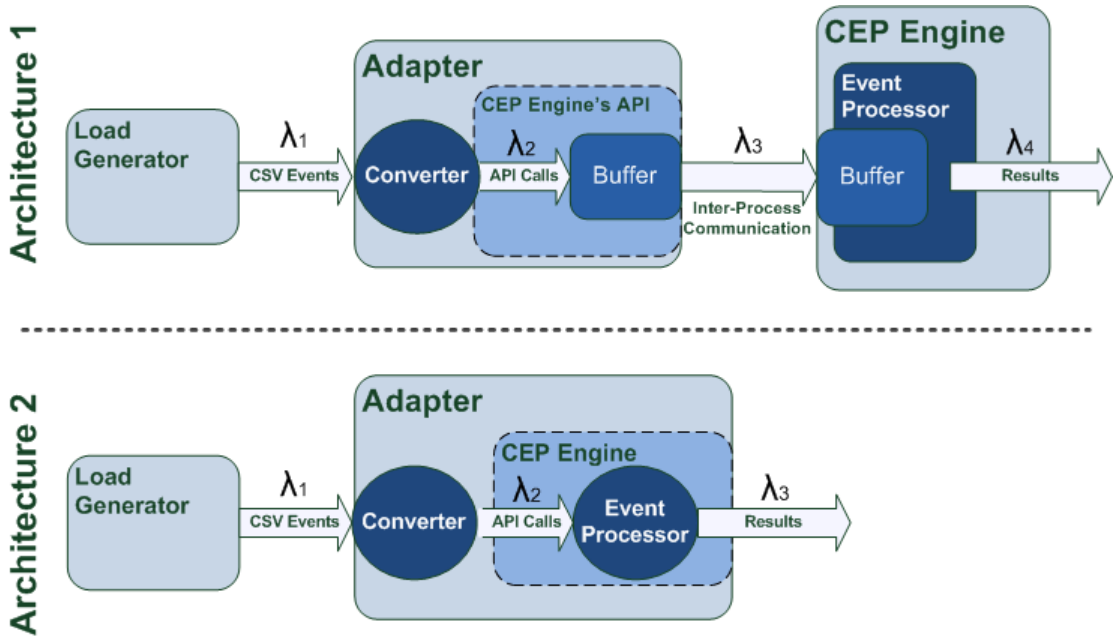


Figure 4.1: Architecture of evaluation setup.

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[5]. The Adapter then converts these messages into the native format of EP engines and transmits them using their respective application programming interfaces (API). The difference between the two architectures shown in Figure 4.1 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.

The input streams data were generated and submitted by the FINCoS framework. 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 EP engines. For all tests, unless otherwise stated, EP engines ran in a single dedicated CPU core, while the load generator and adapters ran in the remaining ones.

4.4.2 Methodology

Tests consisted in running a single continuous query at the EP engine (*except for the multiple-query tests of Section 4.5.6*). 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[6]. 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

[5] Older versions of the FINCoS framework used CSV messages and a dedicated *Adapter* application to isolate the communication with EP systems. Currently, adapters are integrated into *Drivers* and *Sinks*.
[6] The maximum injection rate was determined by running successive tests with increasing throughputs until CPU utilization was maximized or some other bottleneck was reached.

supported by the engines. Tests requiring more time to achieve steady state (*e.g. using long time-based windows*) had a longer duration.

We collected both application-level and system-level metrics. Average throughput and latency were computed by the FINCoS framework. Memory consumption, CPU utilization, and other system metrics were collected using the native *System Monitor* tool of MS-Windows. All the measures reported represent averages of at least two performance runs after the system reaches a steady state.

4.5 Results

In this section we discuss the results obtained after running the microbenchmarks on three EP engines. We emphasize that our primary goal is not to provide an in-depth comparison of existing EP 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).

4.5.1 Selection and Projection Filters

Our first microbenchmark consists in two queries that *filter* rows (*selection*) or columns (*projection*) from a stream of events. The general structure of these queries is illustrated in Figure 4.2 (written in CQL [9]):

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

Figure 4.2: Filtering tests query.

Three parameters, `K`, `N`, and `m`, have their values varied across the several experiments. `K` is used to force the desired selectivity, `N` is the number of input attributes and `m` is the number of projected output attributes (`m≤N`). Two different tests are performed:

i. ***Row selection***: varies predicate selectivity from 1% to 50%; the other parameters are kept constant (`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 results of these two experiments are shown in results in Figure 4.2 below.

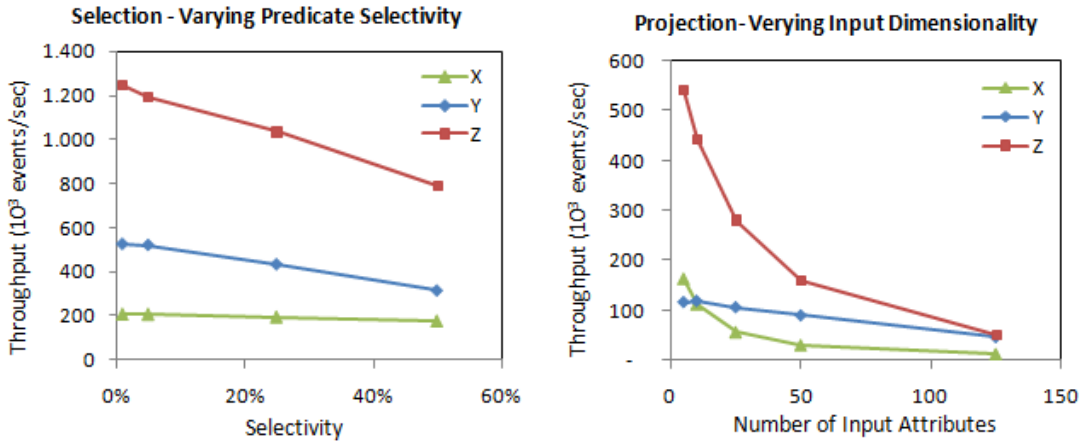


Figure 4.3: Results of filtering tests (selection and projection).

As it can be seen, the throughputs achieved in this test series were very high, measured 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. It should be noted 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 the issue.

4.5.2 Aggregation and Windowing

Count-Based Windows

The second microbenchmark (query `Q2` in Figure 4.4) evaluates aggregations over different count-based window configurations.

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

Figure 4.4: Aggregation over count-based window tests query.

We vary window size (parameter R from 500 to 500K), window type (parameter `S`=1 implies *sliding* window and parameter `S=R` implies *tumbling* 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`). Regarding expiration mode, we expected *sliding* windows to be more expensive than *tumbling* for two reasons. First, *sliding* windows expire tuples one-by-one while *tumbling* windows expire them in batches. Second, *sliding* windows might need to keep more in-memory state (to deal with tuple-by-tuple expirations) while *tumbling* windows may keep only counters and small summary data. Results are summarized in Figure 4.5.

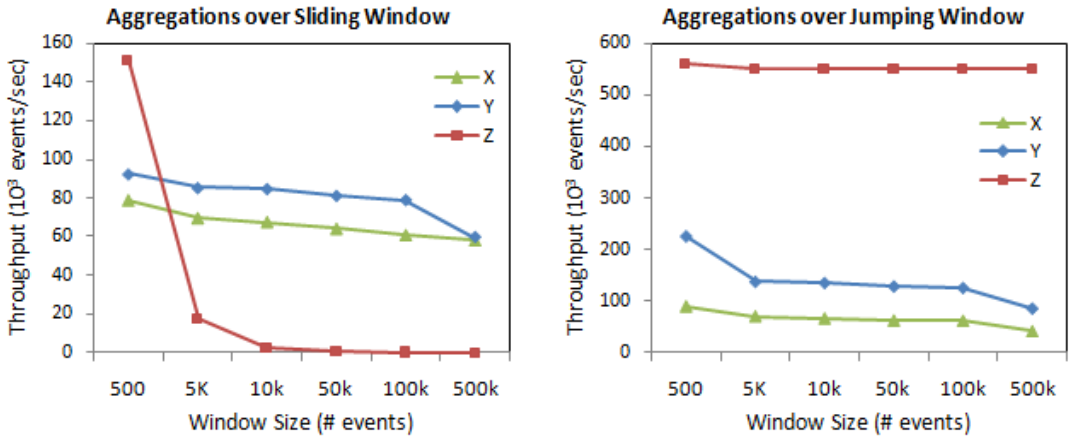


Figure 4.5: Results of aggregation tests: varying windows sizes and policies.

Oddly, engine X had a worse performance with the *tumbling* expiration mode than with *sliding* one. The cause seems to be inefficient batch-expiration of the *tumbling* window tuples as shown by the peak CPU utilization coinciding with the time when the periodic batch-expiration is expected to occur (Figure 4.6).

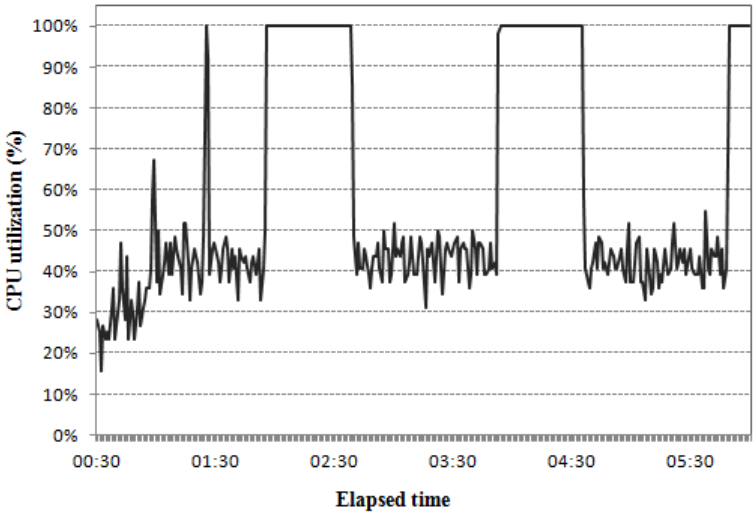


Figure 4.6: CPU utilization of engine X during aggregation test (tumbling window).

On engine Z, the performance difference between the two expiration modes was surprisingly large: very high throughputs with *tumbling* windows (the best of the three

engines at around 550 thousands tuples per 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 appear 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 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, as illustrated in Figure 4.7.

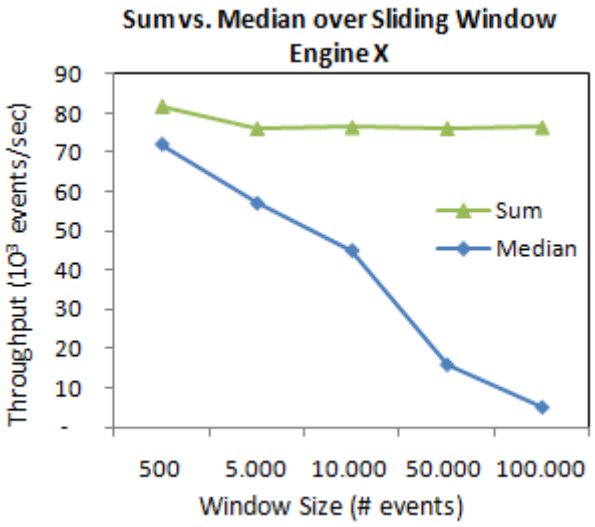


Figure 4.7: Performance of MEDIAN and SUM aggregates.

Time-Based Windows

Large time-based windows over high throughput sources may quickly drain system resources if all incoming events need to be retained. For example, one hour of 20-byte-size events, arriving at a rate of 50k tuples per second, represents around 3.4 GB of space cost. Fortunately, most event-driven applications that compute aggregates require results to be updated only periodically, say every second, rather than for every new event. In those cases, it has been demonstrated [59] that it is possible to compute the aggregation at a much more modest space cost, by pre-aggregating incoming events over a time window of a size equal to the desired update interval, and only then performing the aggregation over a time window with the original size[7]. For example, the aggregation query Q3 can be rewritten into an equivalent, more efficient query Q4, as shown in Figure 4.8 below.

```
Q3: SELECT AVG(A1)
    FROM   A [RANGE 1 HOUR SLIDE 1 SECOND]

Q4: SELECT SUM(s1)/SUM(c1)
    FROM  (SELECT SUM(A1) AS s1, COUNT(A1) AS c1
           FROM A[RANGE 1 SECOND SLIDE 1 SECOND]
          )[RANGE 1 HOUR];
```

Figure 4.8: Two versions of aggregation query over a time-based window.

Q4 computes 1-second aggregates on the inner query and 1-hour aggregates over the 1-second aggregates with the outer query. This results in a significant reduction on memory consumption as depicted next:

[7] Note that this optimization is applicable only for *distributive* and *algebraic* aggregation functions [42].

*Cost of Inner window: (50,000 events/second * 20 bytes/event) * 1 second = 977KB*
*Cost of Outer window: (1 tuple/second * 20 bytes/tuple) * 3,600 seconds = 70KB*

As it can be seen, the optimized version, Q4, has a theoretical space cost of about 1 MB, a three-order of magnitude reduction in comparison with the original cost of 3.4 GB[8].

The microbenchmark of this section consists in running both `Q3` and `Q4`, for different window sizes and varying input rates, with the goal of determining if the tested EP engines are able to automatically perform the aforementioned optimization, and, if not, to quantify, on practice, the performance benefits of employing it. Two distinct experiments were then performed. In the first, input rate was progressively increased while the size of the time window was kept fixed at 10 minutes. The second experiment tested the growth of the queries space cost in the opposite way, by keeping input rate fixed at 100,000 events per second while progressively increasing window size from 20 minutes up to 12 hours. Results are displayed in Table 4.3 and Table 4.4.

Table 4.3: Memory consumption (MB) of the tested engines (10-minute window).

		Input Rate (events/sec)			
Engine	**Query**	500	5,000	50,000	100,000
X	`Q3`	187	1,553	Out-of-memory	Out-of-memory
	`Q4`	39	40	64	98
Y	`Q3`	455	3,173	Out-of-memory	Out-of-memory
	`Q4`	139	141	1,610	1,652
Z	`Q3`	56	64	56	55
	`Q4`	69	68	77	91

[8] In fact, depending on the aggregation function being computed, only the tuples in the outer window might need to be maintained, reducing even more the query space cost. Note that the cost of this "*paned*" approach [59] can be significantly impacted by the number of groups, if a grouped aggregation were used instead. We discuss this topic in further details on Chapter 6.

The numbers reveal that the rewritten query version, Q4, indeed reduced memory consumption when compared to the original query, Q3, for engines X and Y. As expected, Q3 showed a near-linear growth with respect to input rate. The results for X and Y not only demonstrate that the two engines do not automatically implement the two-level aggregate optimization but also reveal an excessive usage of memory resources – *for instance, the expected space cost for a input rate of 5,000 event/sec (20-byte events) is about 57 MB, an amount significantly lower than the memory consumption observed in the tests with those two engines*. As a consequence, engines X and Y ended up exhausting the available memory (more than 13GB) for input rates above 50k events/sec, even on a relatively small 10-minute window. Interestingly, engine Z had its memory consumption only slightly affected by the input rate, with almost identical values for both query versions. These results suggested at first that Z could be the only engine employing the optimization or a similar one.

We then ran the second series of experiments, with much larger windows. The durations of these tests were always 1.5 times the window size. For engines X and Y, we ran the tests only with the optimized version, Q4, since they could not finish the tests using the original query, due to out-of-memory failures. For engine Z we tested both versions, Q3 and Q4.Table 4.4 summarizes the results.

Table 4.4: Memory consumption (MB) of the engines for large time-based windows.

		Window Size				
Engine	**Query**	20 min	1 hr.	2 hrs.	6 hrs.	12 hrs.
X	Q4	114	128	141	146	147
Y	Q4	5,275	5,303	5,232	5,362	5,279
Z	Q3	70	73	55	58	52
	Q4	63	58	46	48	48

As expected, memory consumption remained very stable for all engines when using the rewritten query version Q4, in spite of the window size being increased by a factor of

36. These results confirm that the two-level aggregate optimization is very effective in reducing resource consumption for aggregations over large temporal windows.

More importantly, however, the new experiments exposed a behavior of engine Z not revealed in previous tests. While in the first experiments Z was 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 impacted by the window size. As shown in Figure 4.9, Q3 had a drastic drop in maximum throughput as window size was increased, while Q4 maintained a very steady throughput curve.

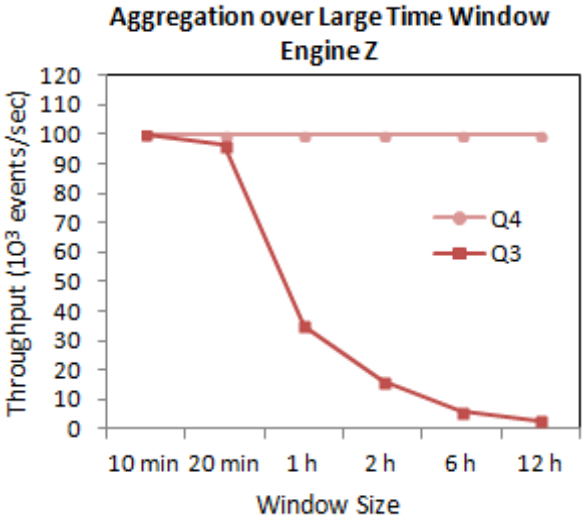


Figure 4.9: Results of the aggregations over large time windows test (engine Z).

While in the tests with Q3 CPU was always pushed to its maximum (*for windows of 20 min and beyond*), with Q4 CPU utilization stayed always around 1%. These numbers indicate that Z also does not perform the optimization mentioned earlier, and employ instead an alternative implementation strategy that sacrifices query throughput to keep memory consumption controlled.

4.5.3 Joins

The third series of tests evaluated the *join* performance of EP engines. There are numerous factors that affect performance of joins such as the input source type (e.g., stream, window, or table), number of elements in each input, join selectivity (i.e., number of results generated per input event), cardinality (number of unique values) of the join attribute, number of joining sources, join types, and throughputs of incoming streams. In this microbenchmark we focus on equi-joins. Three test series are defined, each with different data sources and factors under analysis:

J1. ***Window-to-window***: joins two sliding 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 in an in-memory table;

J3. ***Stream-to-DBMS-relation***: joins events with data stored in a table of an external database.

Query definitions and results of the tests above are presented next.

J1: Window-to-window

The query for the window-to-window join test is shown Figure 4.10 below.

```
Q5: SELECT *
    FROM   stream1 [ROWS S SLIDE 1] AS S1,
           stream2 [ROWS S SLIDE 1] AS S2
    WHERE  S1.ID = S2.ID
```

Figure 4.10: Window-to-window join tests query.

Q5 joins the contents of two sliding windows of size “S” defined over two distinct streams, using the attribute “ID” of each stream as correlation criteria. We then define two tests with the objective of examining how window size and join selectivity affect the query performance:

J1-1 ***Varying window size and keeping join selectivity fixed***: parameters *S* and *MAX_ID* take the same values, from 500, to 500k, which ensures a constant 100% join selectivity (each input event finds one and only one match on the other window);

J1-2 ***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 4.11 below shows the results for this test series.

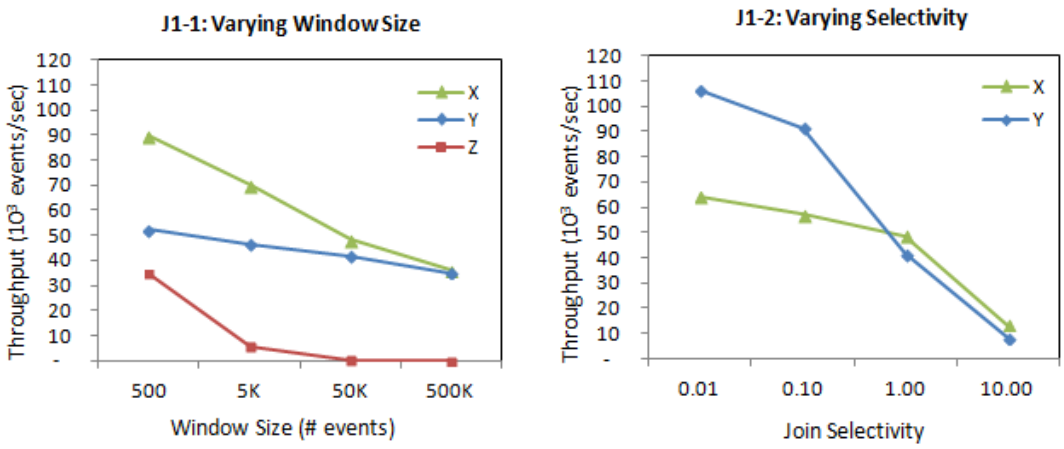


Figure 4.11: Results of window-to-window join tests.

The tests 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 one. Once more engine Z had an acute drop on query throughput as the window size was increased, similarly to what happened in the aggregation tests of section 4.5.2. In order to minimize the cost of window maintenance, and thus be able to observe the effect of selectivity on Z, we ran a modified version of J1-2, with a smaller window (size 500, not shown in the graph). However, there were no noticeable performance differences when varying the join selectivity, showing again that sliding windows are not efficiently handled by Z, and for this reason dominate query cost.

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

We now discuss the results for the stream enrichment tests (J2 and J3). The queries for those experiments have the format shown in Figure 4.12.

```
Q6:  SELECT *
     FROM   stream1 AS S,
            table1  AS T
     WHERE  S.ID = T.ID
```

Figure 4.12: Stream-to-table join tests query.

In both tests an event stream "S" with 4 fields is joined with a static table "T" with 10 fields. In J2 the EP 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; we tested both with MS-SQL Server™ 2005 and Oracle™ 11g, and the results were similar*). The number of records in the table ranged from 1k to 10M for J2 (in-memory), and from 1k to 100M for J3 (DB). The join selectivity in all tests is 100% (i.e., every event in the stream is matched against one and only one record in the table). Figure 4.13 shows the corresponding results.

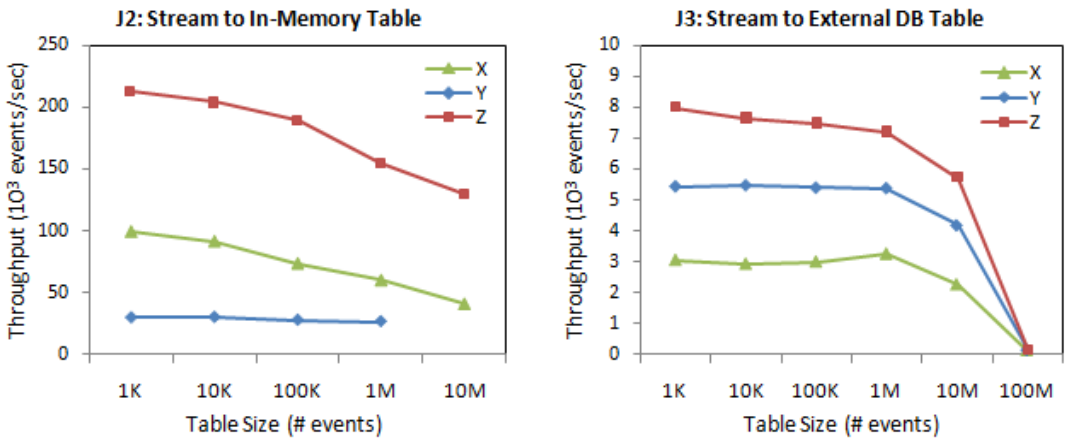


Figure 4.13: Results of stream-to-table join tests.

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 observe how engine Z had a considerably better join performance when operating over a table rather than over sliding windows (*see J1-1, in Figure 4.11*).

In the J3 experiment series, two facts are worth mentioning: first, neither the EP 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 I/O, resulting from disk requests, significantly lowered the query throughput, as it can be seen in Figure 4.13.

4.5.4 Pattern Matching

Event pattern matching was exercised using queries with the structure of query Q7 below. Q7 searches for instances of two related events (i.e., with the same "id"), happening within a time-window of size *interval*, where the "A1" attribute of the second event is above some threshold *K*.

```
Q7:  SELECT *
     PATTERN SEQ(A a1, A a2)
     WHERE   a1.id = a2.id
             AND
             a2.A1 > K
     WITHIN  interval
```

Figure 4.14: Pattern matching tests queries (expressed using SASE+ language).

The consumption policy used in the tests was always the "*all-to-all*", the only supported and semantically equivalent across all the tested EP engines. The purpose of the "`a2.A1>K`" predicate is to verify that EP engines indeed benefit of predicates in pattern detection by pushing them earlier in query plan construction. We then examine the effect of three factors:

i. ***Window size***: we vary parameter *interval* from 10 up to 600 seconds. The other parameters are kept fixed (*MAX_ID*:10k and *K* ensures a selectivity of 0.1%);

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

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

Figure 4.15 show the results for the three test series. Interstingly, in the first experiment, all the engines had a very similar decrease in throughput as *interval* got larger (a total drop of 50% for engine X and 54% for engine Y). 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 1.5 GB of memory*). Considering only the values of *interval* parameter between 10 and 120, throughput in Z dropped 28%, 24% in engine X, and 29% in engine Y.

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 – the results of this last test series indicate that all engines indeed employ the predicate push-down optimization.

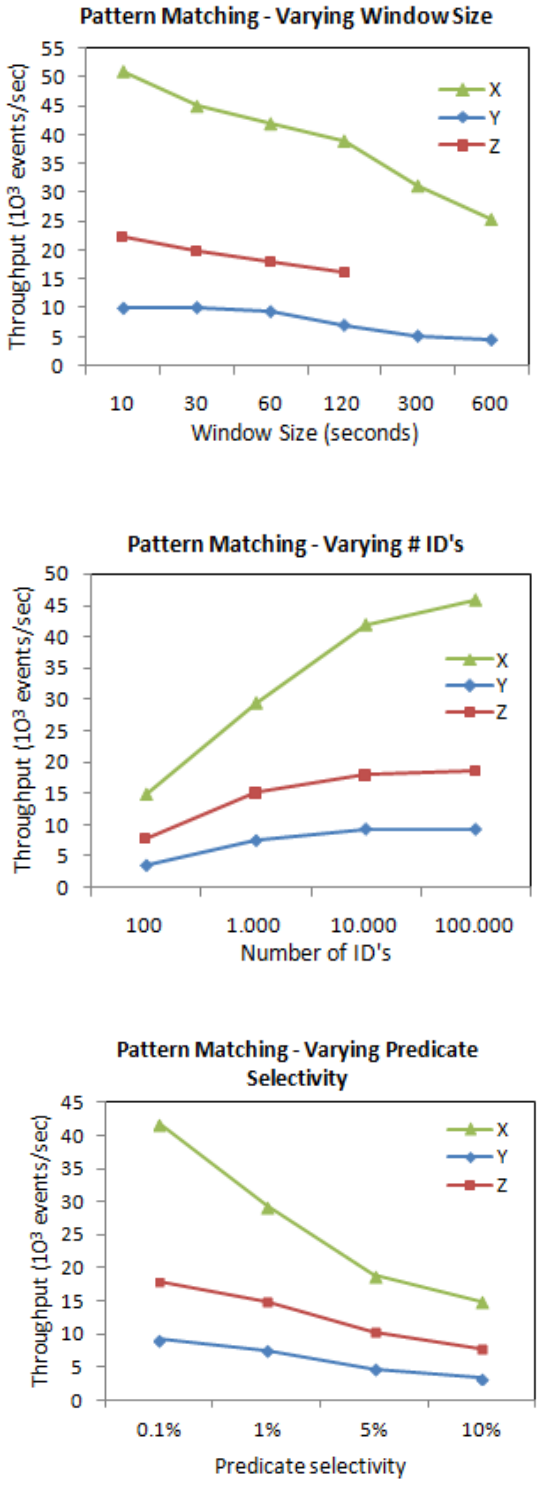


Figure 4.15: Results of the pattern matching tests.

4.5.5 Adaptivity to Bursts

Different from the previous tests, which stressed a particular basic event processing operation, this microbenchmark has the objective of assessing how fast and efficiently the tested 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. In order to evaluate the adaptivity of engines to bursts on event arrivals, we arrange each experiment in four distinct parts (*see Figure 4.16*):

- 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%;
- A 5-minute *steady* phase during which the injection rate is kept fixed at λ;
- A 10-second *peak* phase during which the injection rate is increased 50% (to 1.5λ), making the system temporarily overloaded;
- A 5-minute *recovery* phase in which the injection rate is again fixed at λ.

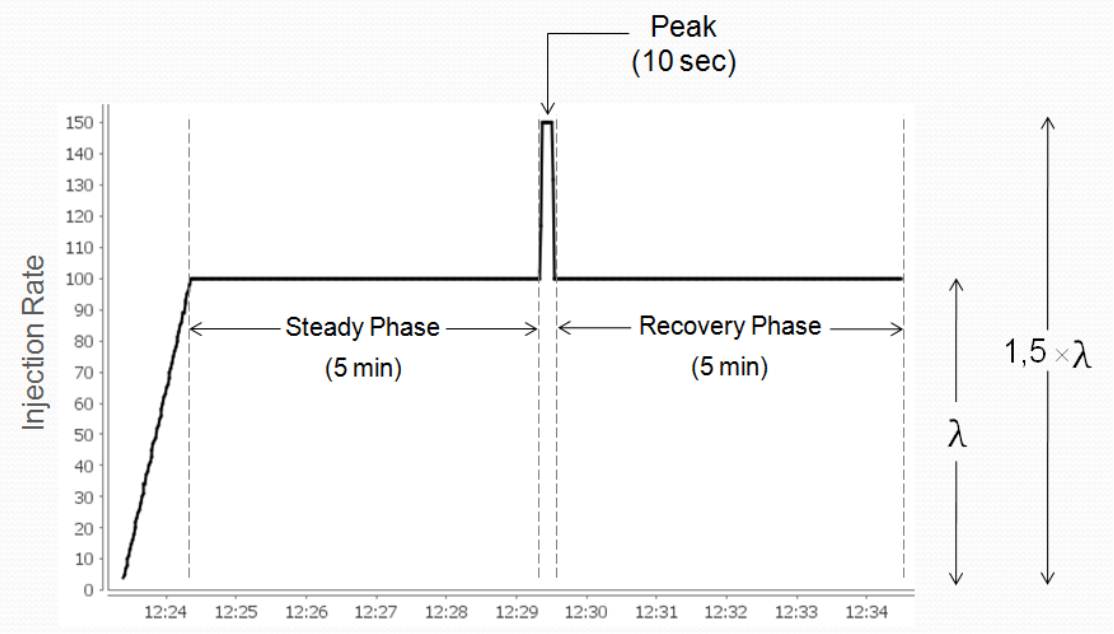


Figure 4.16: Adaptivity test.

We then define the following metrics in order to characterize the adaptivity of the event processing systems:

- ***Maximum peak latency***: the maximum observed response time either during or after the injection of the peak load.

- ***Peak latency degradation ratio***: quantifies the increase in latency caused by the peak. It is computed as the ratio between the 99.9th-percentile latency of peak phase with respect to 99.9th-percentile latency of steady phase:

$$\frac{99.9^{th}\ latency_{peak}}{99.9^{th}\ latency_{steady}}$$

- ***Recovery Time***: measures how long it takes for the system to return to the latency levels of the steady phase after the peak in the load is interrupted. Algebraically:

$$\tau_{\text{recovery}} - \tau_{\text{peak}}$$

  where τ_{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.

- ***Post-peak latency variation ratio***: measures the variation of average latency after recovery in comparison with the average latency during steady phase. The purpose of this metric is to determine if the systems return, after the peak, to a similar state to the one they were before the peak. It is computed as follows:

$$\frac{Avg.latency_{after\ recovery}}{Avg.latency_{steady\ phase}}$$

The workload of the adaptivity tests consisted in the aggregation query Q2, from section 4.5.2.

Discussion: Blocking/Non-Blocking API and Latency Measurement

Recall from Figure 4.1 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 overloaded while Y and Z prevent clients from submitting events at a higher rate than that they can process. As discussed in section 3.4, there are multiple ways of computing latency. In order to properly measure latency for blocking calls, it is necessary to employ the *scheduled time* of input events instead of their *send time* – formula (3) in Figure 3.4. This way it is possible to account 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.

Results

Table 4.5 and Figure 4.17 summarize the results of the adaptivity tests. As it can be seen, engine X, which adopts a non-blocking approach 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. After recovery, though, all engines returned to virtually the same latency level as that observed before the peak.

Table 4.5: Results of Adaptivity Tests.

	Engine		
Metric	X	Y	Z
Maximum Peak Latency (ms)	4,725.0	1,262.0	1,483.0
Peak latency degradation ratio	82.8	57.4	5.9
Recovery time (seconds)	43.1	1.3	1.6
Post-peak latency variation ratio	1.0	0.9	1.0

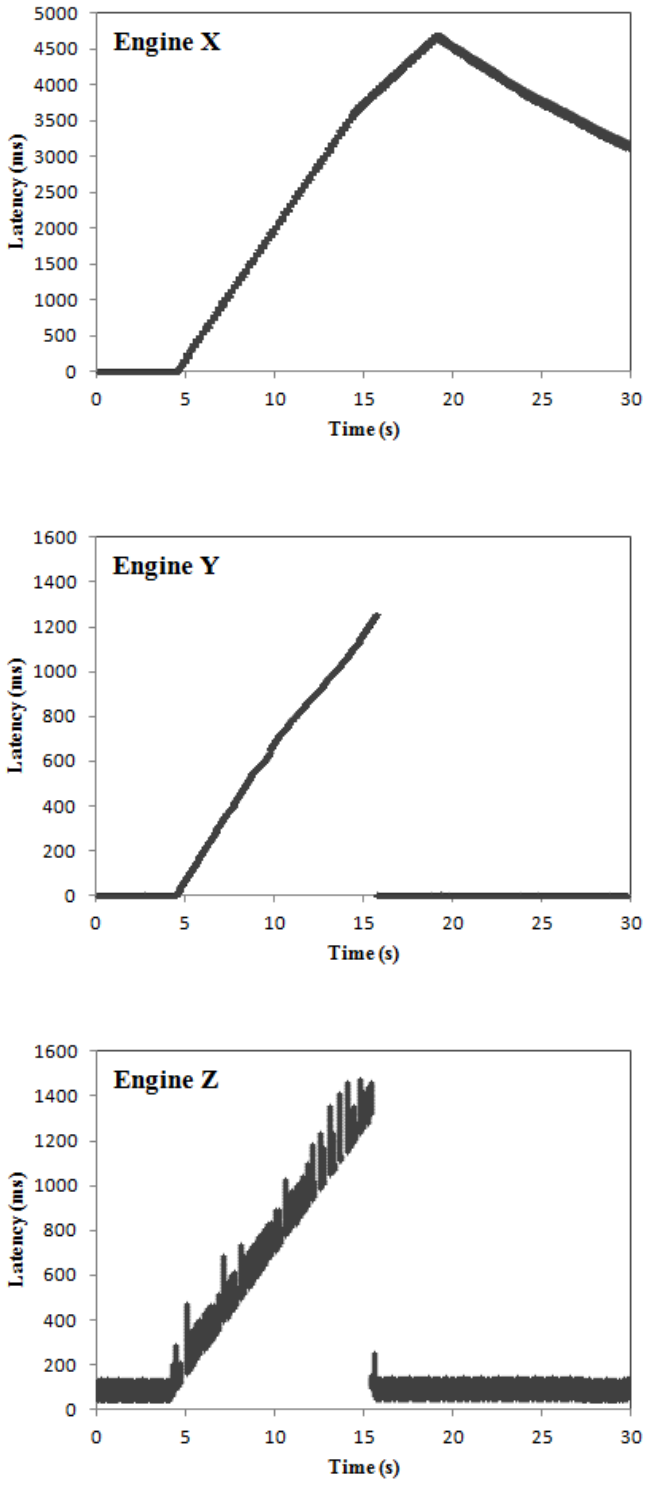


Figure 4.17: Scatter plot of latency before, during, and shortly after the peak.

4.5.6 Multiple Queries and Resource Sharing

The objective of this microbenchmark is to analyze how the EP engines scale with respect to the number of simultaneous similar queries, and if they perform some form of resource sharing. The query used in this experiment is a window-to-window join, like Q5 of section 4.5.3. We tested two variations:

- **Test 1: Identical queries**. In this test we focus on computation sharing and the main metric is 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.

In both experiments, the number of concurrent queries was progressively increased, assuming the values N= {1, 4, 16, 64}. Results are shown in Figure 4.18 below.

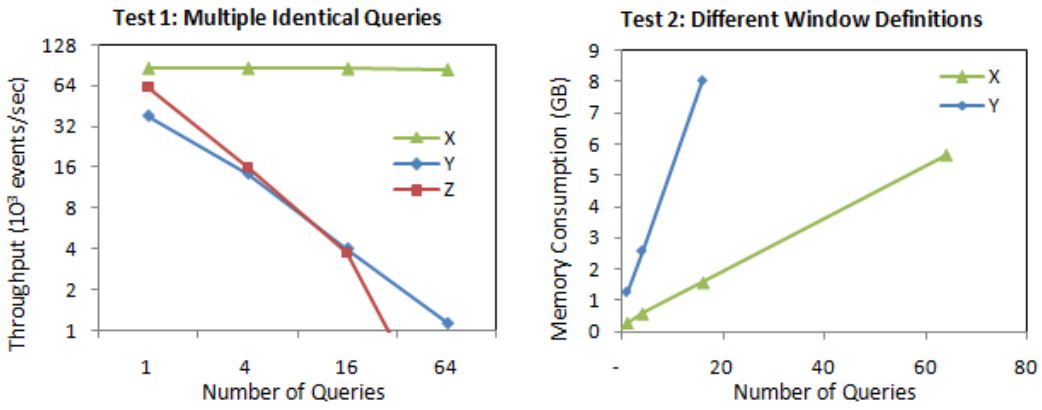


Figure 4.18: Results of scalability tests.

The numbers indicate that neither Y nor Z implement any query plan sharing mechanism. In the first test both engines had their throughput dropping linearly with the

number of simultaneous queries (in spite of those performing exactly the same computation). Overall, engine X showed to be the only one to implement some kind of query plan sharing – its throughput remained unaffected when the number of queries was increased in the first test.

However, in the second test, in which queries were similar but different, engine X was not able share resources, and memory consumption increased linearly with the number of simultaneous queries. The same happened with memory consumption of Y in the second test – as a consequence, it ran out of memory in the experiment with 64 queries. We could not determine memory consumption for engine Z in the second test because it became unresponsive while the window was being filled.

4.6 Related Work

There have been some previous efforts in characterizing the performance of event processing platforms. Dekkers [27] conducted a study for evaluating event pattern detection performance of two open-source EP engines: Esper [34] and Streamcruncher [95]. The tests consisted in sending a progressively larger number of synthetic event patterns (with and without noise events between their component events) and measuring how long the engines take to process them. The results of this study showed that Esper outperformed Streamcruncher in all tests.

A performance study of the Bea WebLogic Event Server engine (now Oracle OEP) is presented by White et al. [103]. The tests simulate a simple financial trading application, which monitors the price movements of a number stock symbols. The workload consists in two continuous queries, replicated for each of the 200 symbols being monitored, on a total of 400 queries. Injection rate ranged from 100,000 up to 1,000,000 events per second. The results of the study demonstrate the engine's ability in handling very large volumes of events while providing deterministic latencies. It should

be noted, though, that the queries used are very simple, especially regarding memory cost (the engine keeps sliding windows of only two or three tuples).

The STAC benchmark council [82] disclosed a report measuring the performance of the Aleri Event Stream Processor (now incorporated into the SAP-Sybase portfolio [97]) in consolidating order book data from two high-volume feeds, fed through a Reuters Market Data System (RMDS) [86]. The objective was to measure the total latency introduced by the Aleri solution when running on RMDS. The results demonstrated very low latencies (less than 3 milliseconds) for input rates of up to 180,000 order book updates per second. The report, however, did not disclose sufficient details about the test case that allowed fully understanding how complex the scenario was or replicating the results.

The open-source engine Esper is distributed with a toolkit for evaluating its performance. The tests simulate a simple stock market application and consist in computing the volume-weighted average price (VWAP) of 1,000 symbols. A report with results using this kit can be found in [33].

It is worth noting that all the aforementioned work and tools touch only part of the event processing functionality. To the best of our knowledge, the study presented in this chapter is the first to exercise the wide range of functions provided by an event processing platform. A more recent work by Dayarathna and Suzumura [26] follows this comprehensive approach, and present a study comparing the performance of the stream processing engines System S, S4 and Esper using a number of application scenarios and microbenchmarks.

4.7 Summary

In this chapter we presented an extensive performance study of event processing systems. We proposed a series of microbenchmarks to exercise the core event processing operations and then carried out experimental evaluations on three EP platforms. The tests confirmed that very high throughputs can be achieved by EP 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 tumbling and sliding windows was about 4 orders of magnitude. Aggregation tests also revealed a poor utilization of memory resources by two of the three tested systems. Further, the well-known optimization of pre-aggregating data when computing periodic sliding-window aggregates was not implemented by any engine. With respect to joins, tests revealed that accessing data stored in databases can significantly lower system throughput. Pre-loading static data into EP engine offers good performance and may thus solve the issue, but this approach is feasible only when data does not change often and fits in main memory. We also concluded that the tested engines had very disparate adaptivity characteristics, with the approach used to receive events from clients – *either blocking or non-blocking* – apparently playing a fundamental role on that aspect. Finally, the tests with multiple queries showed that plan sharing happened only in one EP engine and only for identical queries. We also note that the EP engines were not able to automatically benefit from the multi-core hardware used in our tests. In general terms, we concluded that no EP engine showed to be superior in all test scenarios, and that there is plenty of room for performance improvements. We dedicate the next two chapters of this dissertation to this goal, by investigating and proposing ways to enhance efficiency and scalability of EP systems.

Chapter 5

Performance Enhancements for EP Systems - Part I: CPU and Memory

The study conducted in the last chapter revealed important performance issues on the current generation of event processing platforms. In particular, we observed that some systems grossly waste resources and that their performance tends to drop significantly as query state increases. In this chapter we further quantify some of these inefficiencies, identify their causes, and propose changes on internal data structures and cache-aware algorithms to overcome them. We test the before and after system both at the application and microarchitecture level and show that: i) the changes improve microarchitecture metrics such as clocks-per-instruction, cache misses or TLB misses; ii) and that some of these improvements result in very high application level improvements such as a 44% improvement on stream-to-table joins with 6-fold reduction on memory consumption, and order-of-magnitude increase on throughput for moving aggregation operations.

5.1 Motivation and Related Work

Previous work by Ailamaki [4], Ramamurthy [77], and Abadi [2] showed that microarchitecture inspired improvements such as cache-aware algorithms and changes of internal data representations can lead to high improvements on the performance of conventional data management systems. Encouraged by this work, we took a similar

position and set-out to discover the microarchitecture performance of event processing systems.

Using Esper [34], a widely used open-source EP engine, we measured the system performance executing common operations such as moving aggregations and stream-to-table joins. We monitored the system at the microarchitecture level using the *Intel VTune*® profiler [51], and also collected application-level metrics such as memory consumption and peak sustained throughput.

To isolate from secondary effects, we then replicated the main algorithms and data structures on our own event processing prototype and progressively improved them with microarchitecture-aware optimizations. These optimizations were then validated first by running the tuned prototype in a multi-query scenario, and then porting the modifications back into Esper.

5.1.1 Summary of Contributions

The main contributions of this chapter are the following:

- We analyze how current event processing systems perform at both application and hardware levels. By collecting and correlating metrics such as throughput, CPI, and cache misses during execution of continuous queries, we show that microarchitectural aspects significantly influence the final performance of common event processing tasks, being in some cases the sole cause for performance degradation when the input is scaled up (Section 5.3).
- We demonstrate how alternative data structures can drastically improve performance and reduce resource consumption in EP systems (Section 5.4).
- We implement, test and evaluate an adapted version of the Grace Hash algorithm [55] for joining event streams with memory-resident tables. Results

revealed that by reducing the impact of microarchitectural aspects on query execution performance was improved by up to 44% (Section 5.5).

- We propose, implement, test and evaluate a microarchitecture-aware algorithm for computing moving aggregations over sliding windows. In our experimentation evaluation the proposed optimization provided performance gains ranging from 28% to 35% (Section 5.5).

5.2 Background

From the microarchitectural point of view, the amount of time a given computational task *T* takes to complete depends primarily on two factors: the task size, measured in number of instructions (a.k.a. *instruction count* or *IC*), and the average duration of instructions (frequently expressed as *cycles per instruction* or *CPI*). Algebraically, in number of cycles [47]:

$$CPU\ execution\ time = IC \times CPI$$

Better performance can be achieved by reducing either factor or both. Traditionally, software developers have focused on reducing IC by improving time complexity of algorithms, but an increased interest in making a more efficient use of hardware resources has been observed over the last years [4][5][90][107].

To understand how these optimizations targeted at the hardware level work, it is necessary to know the internals of CPU operation. Every single instruction is executed inside the processor as series of sequential steps across its several functional units. During this sequence of steps, generally referred as *pipeline*, CPU instructions are fetched from memory, decoded, executed and finally have their results stored back into registers or memory. To increase throughput, instructions in different stages/functional units are processed in parallel (*Instruction-Level Parallelism*). In ideal conditions, the

processor pipeline remains full most of the time, retiring one or more instructions per cycle (implying CPI ≤ 1).

Many factors, however, can cause instructions to *stall*, thus increasing average CPI. The gap between processor and memory speeds is one of them: an instruction that access data resident in main memory may require tens to hundreds of CPU cycles to complete. Another typical cause of stalls is *data dependency*: when an instruction *j* depends on a value produced by an instruction *i* that was fetched closely before it, *j* cannot execute until *i* completes (in fact, if the processor issues instructions in program order, when instruction *j* stalls, no later instructions can proceed, thus aggravating the performance impact of the data dependency). Finally, *control dependencies*, which happen when the instruction flow cannot be determined until a given instruction *i* (e.g., a conditional branch) completes, can also adversely affect the degree of instruction-level parallelism achieved.

In order to attenuate the aforementioned stalls, hardware vendors have devised several techniques. For example, to minimize memory-related stalls, smaller and faster *cache* memories are placed in the data path between processor and main memory. The strategy is to benefit from the locality principle and serve most memory requests with data coming from lower-latency cache accesses. Additionally, data dependencies are minimized by allowing instructions to execute out-of-order inside the pipeline. Finally, control dependencies are partially addressed via speculative execution (i.e., the processor executes instructions that lie beyond a conditional branch as if it had been already resolved).

In practice, the characteristics of the applications determine whether the hardware techniques above will be successful or not at making the processor execute close to its full capacity. With that in mind, a number of novel database systems had been developed over the last years, in an attempt to better exploit the internal features of processors [90] [107]. Examples of microarchitectural optimizations employed by such databases include a column-oriented data organization and compression techniques

which together provide a more efficient use of memory hierarchy (e.g., [90] and [107]). Also, recently proposed compression algorithms [108] minimize the negative impact of branch mispredictions by avoiding `if-then-else` constructs in their critical path.

In this chapter, we argue that similar optimizations can be applied in the context of event processing systems. In fact, our expectation is that those microarchitectural improvements result in potentially higher gains as, contrary to databases which manipulate data from secondary media, most of the data processed by EP systems resides in main memory.

5.3 Performance Analysis

In order to validate that hypothesis, we conduct a preliminary performance analysis of the Esper event processing platform. Our goal is to assess how well the engine uses the available resources and verify if microarchitectural aspects indeed play a significant role on the overall system performance. In the rest of this section we introduce the workload and methodology used to stress the Esper engine and discuss the results of this preliminary analysis. Later in this chapter, we use the same test case to validate the optimizations proposed in the following sections.

5.3.1 Test Case

Similarly to the study presented in Chapter 4, we use common core operations performed by event processing systems to assess the engine under test. The workload here consists in processing either:

i. A moving *aggregation* over a windowed event stream or

ii. A *join* of an event stream with historic data.

These two queries and the dataset are described in detail next.

Dataset

Input data for the experiments of this chapter consisted in a generic event stream *S* and table *T* with schemas shown in Figure 5.1 and Figure 5.2.

```
STREAM S (
      ID integer,
      A1 double,
      A2 double,
      TS long
      )
```

Figure 5.1: Schema of input event stream "S".

In our tests the values assumed by `S` attributes are not relevant for query performance evaluation, so they were filled with a fixed, pre-generated, data value – *this ensures that measurements are minimally affected by data generation*. The exception is the attribute `ID`, which is used to join stream `S` with table `T`. In this case, `S`'s `ID` was filled with random values uniformly distributed in the range of `T`'s `ID`.

```
TABLE T (
      ID integer,
      T1 integer,
      T2 integer,
      T3 integer,
      T4 integer
      )
```

Figure 5.2: Schema of historical table "T".

The `ID` attribute of table `T`, used to perform the join with the event stream, assumes unique values ranging from 1 to `TABLE_SIZE_IN_ROWS` (a parameter that changes from one experiment to another). The other four attributes in `T` do not influence query performance and are filled with random data.

Aggregation Query

The aggregation query used in this test case computes a moving average over a *count-based sliding* window. The query specification is shown next using the syntax of the CQL language [9]:

```
SELECT AVG(A1)
FROM   A [ROWS N SLIDE 1]
```

Figure 5.3: Aggregation query.

For this particular query, every event arrival at stream `S` causes the output of an updated result. Parameter `N` represents the window size, which varies across the tests, ranging from 1,000 to 100 million events.

Join Query

To examine the behavior of the EP system under test when performing a join, we used a query based on a real use-case of a telecom company that needed to join streaming call detail records (CDR) (here represented by stream `S`) with historic data (represented by table `T`). In our tests this query is expressed as follows:

```
SELECT  S.ID, S.A1, T.T1
FROM    S, T
WHERE   S.ID = T.ID
```

Figure 5.4: Join query.

Since the goal here is to focus on the performance of processor and memory hierarchy, the table is maintained in main memory, thus eliminating eventual effects of the I/O

subsystem on the results[9]. The selectivity of the query is always 100% (i.e., every event is matched against one and only one record in the table) and the table size is varied across tests, ranging from 100 to 10 million rows.

5.3.2 Setup and Methodology

All the tests were carried out on a server with two Intel Xeon E5420 Quad-Core processors (Core® microarchitecture, L2-Cache: 12MB, 2.50 GHz, 1333 MHz FSB), 16 GB of RAM, running Windows Server 2008 x64 and Sun Hotspot x64 JVM.

The performance measurements were done as follows:

- A single Java application was responsible for generating, submitting and consuming tuples during the performance runs. Events are submitted and processed through local method calls, so that measurements are not affected by network/communication effects.
- In the tests with join queries, load generation was preceded by an initial loading phase, during which the in-memory table was populated with a given number of records.
- Load generation started with an initial 1-minute warm-up phase, with events from stream `S` being generated and consumed at the maximum rate supported.
- Warm-up was followed by a 15-minute measurement phase, during which we collected both application-level metrics and hardware-level metrics. Application-level metrics, namely throughput and memory consumption, were gathered inside the Java application. Throughput was computed as total event

[9] As pointed out before, keeping the dataset in memory is commonplace in most EP applications, especially those which require high processing throughputs and/or low latencies.

count divided by elapsed time. Memory consumption was computed using `totalMemory()` and `freeMemory()`, standard Java SDK `Runtime` class methods. Hardware-level metrics were obtained using the *Intel VTune*® profiler [51], as described in detail next. As in the warm-up phase, events were submitted at the maximum rate sustained by the specific implementation.

- Each test was repeated 3 times and the reported metrics were averaged.

VTune collects metrics by inspecting specific hardware counters provided by Intel processors. To avoid excessive monitoring overhead, only one or two events are collected at a time – for doing that, VTune breaks event collection in "runs". In our tests, each run has a duration of 2 minutes (1 minute for calibration and 1 minute for counters collection), and 6 runs were necessary to collect all the configured metrics.

5.3.3 Results

Figure 5.5 illustrates the results for the join query running at the event processing system Esper, with two different tuple representations used by the engine: *Map* and *POJO* (in the former events are represented as instances of the standard `HashMap` Java class, while in the latter events are represented as fixed-schema Plain Java Objects) [10].

Two important observations can be made from the results shown in Figure 5.5. First, for both tuple representations, the throughput dropped about 40% from a join with a 1,000 rows table to a join with a 10-million rows table. Note that this drop occurred even though the employed algorithm (i.e., hash join) has a theoretical O(1) runtime complexity. The collected metrics indicate that this behavior was mainly due to

[10] In this chapter we focus on Esper due to its open-source nature, which allowed us to port some of the optimizations here proposed into a real engine. Still, the results here presented are representative of event processing engines in general, given that the behavior exhibited by Esper is similar to what was observed in previous chapter with all the other tested systems.

microarchitectural aspects, and to a lesser extent, to increased garbage collection activity.

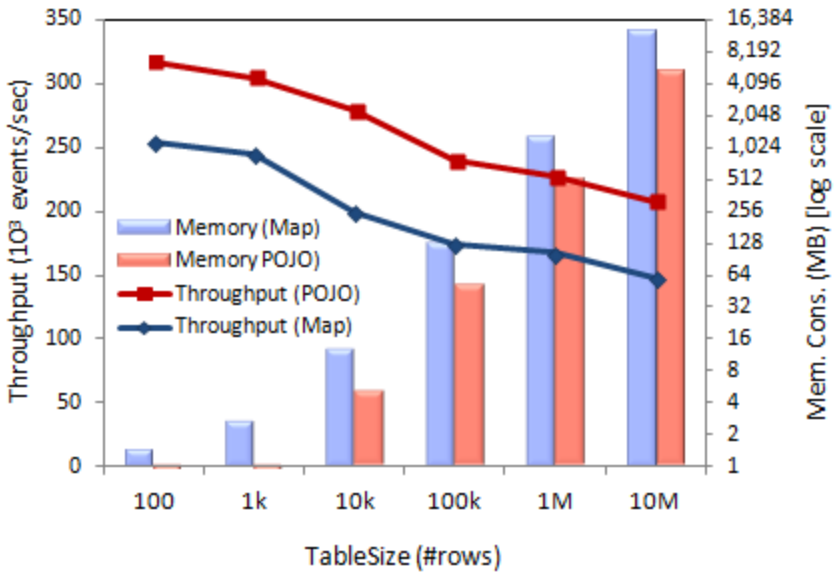


Figure 5.5: Performance of join query on the Esper EP engine.

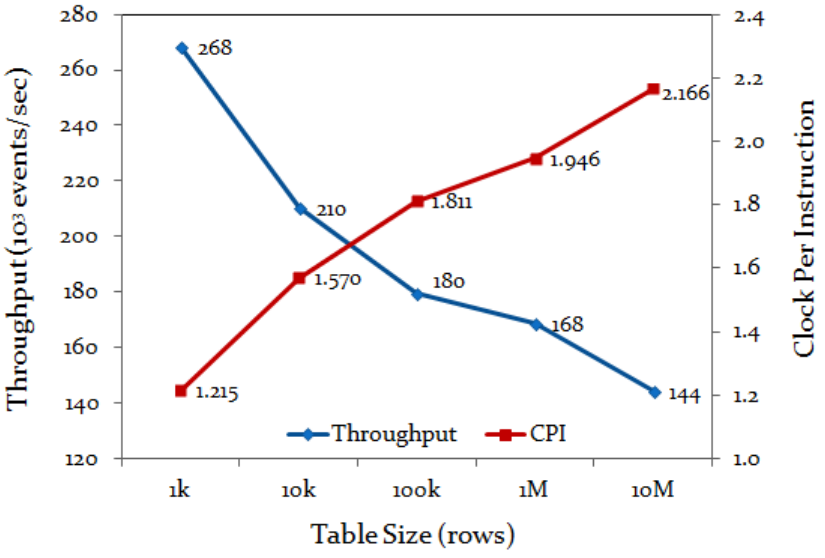


Figure 5.6: Relation between application performance and hardware metrics.

As it can be seen in Figure 5.6, query throughput decreased at a similar proportion – 86% drop from 1k to 10M rows – to the increase observed in the CPI metric – about 78% from 1k rows to 10M rows. In the same interval, the time spent with garbage collection increased from 0.01% to 1.90%, which justifies the slightly larger drop on application throughput than the performance degradation at the hardware level.

The second point worth mentioning is that measured memory consumption during query execution was 4 to 60 times the space strictly required for keeping the table data. Examining Esper's source code we concluded that this excessive memory consumption was caused by non-optimized internal representation of tuples as further explored in the next section.

5.4 Optimizing Data Structures

In order to address the problem of excessive memory consumption, we focused first in optimizing the structures used to keep data items in main memory (i.e., the window for the aggregation query and the table for the join query). Specifically, we were interested in finding out if the original representations used by Esper to represent the events and table tuples could be improved and if a column-oriented storage model would result in enhanced performance in the context of event processing.

It is worthy to notice that column-store formats are especially useful for *read-intensive*, *scan-oriented*, *non-ad-hoc* queries. Thus, while on one hand EP systems, with their scan-oriented, non-ad-hoc queries may benefit from a column-oriented storage model, on the other hand, the read/write nature of those queries might reduce, or even eliminate, the eventual gains obtained with this alternate data organization scheme.

To assess the impact of data structures on query performance, we started representing events/tuples as instances of the `HashMap` class – Figure 5.7 (a) – and then employed progressively more lightweight representations: first arrays of Objects (b), and then

fixed-schema Plain Java Objects (POJO) (c). Finally we tested the column-oriented storage model (d), in two different modalities: first keeping all original attributes of events/tuples (here named "*Col-Store*") and then keeping (projecting) only the attribute referenced in the query ("*Col-Store Proj.*"). In the *Col-Store* format, N aligned arrays of primitive types are kept in memory (where "N" is the number of attributes of the incoming events), while in the *Col-Store Proj* format only one array containing the values for the referenced attribute is maintained.

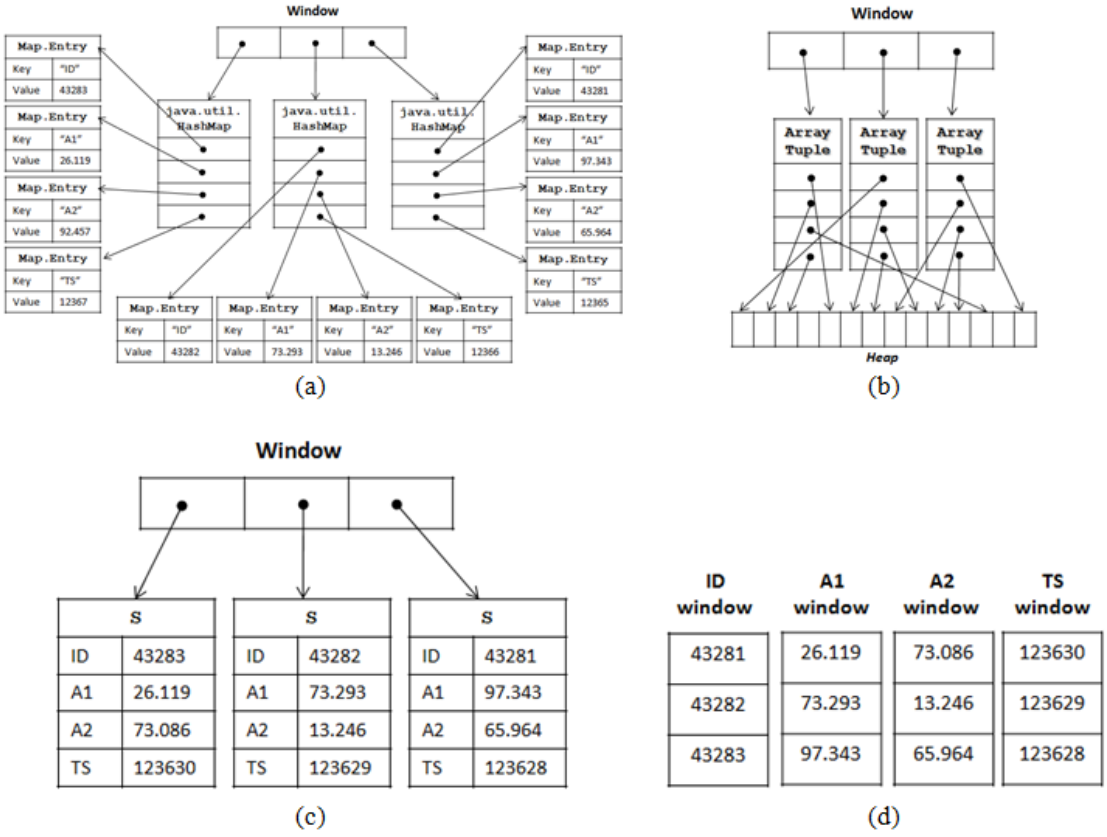


Figure 5.7: The different data structures used to represent tuples.
(a) Maps; (b) Object array; (c) Plain Objects; (d) Column-store

We tested each tuple representation with both the aggregation and the join queries, using our own event processing engine prototype – *we used a neutral small prototype to avoid that the evaluation became tied to a particular product and prevent results from being affected by any specificities of the Esper engine*. The algorithm for computing the aggregation query works as follows. The sliding window is implemented as a circular buffer, internally represented as a fixed-length array; the average aggregation itself is computed by updating count and sum state variables upon event arrival/expiration. The join algorithm is also straightforward: it keeps the table into a hash index structure with the join attribute as key and then performs a lookup in that hash table every time a new event arrives. The results of these tests are summarized in Figure 5.8, Table 5.1, and Table 5.2.

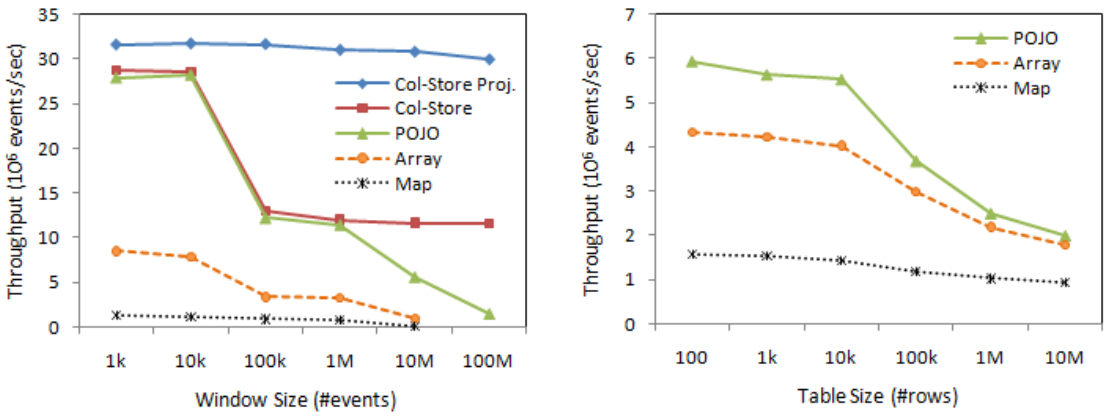


Figure 5.8: Impact of internal representation on performance
(a) aggregation (b) join.

The numbers corroborate that considerable gains in final performance and resource consumption can be obtained by using more lightweight data structures. For the join query, the most efficient representation achieved more than 3 times more throughput than the original one, while consuming less than 15% memory space of what was originally required. The gains for the aggregation query were even more impressive: the

best performing implementation (column-store keeping only required data) achieved on average about 35 times more throughput than the one using maps as event representation and reduced memory consumption to around 1.5% of the memory space occupied by that implementation.

Table 5.1: Data structures and memory consumption (in MB): aggregation query.

Tuple Format	Window Size					
	1k	10k	100k	1M	10M	100M
Map	0.8	5.5	51.4	511.5	5,112.9	-
Array	0.5	2.1	17.9	176.5	1,755.0	-
POJO	0.3	0.8	6.3	61.2	682.1	6,103.0
Col-Store	0.2	0.5	2.8	26.9	267.2	2,670.5
Col-Store Proj.	0.2	0.3	0.9	7.8	76.0	763.1

Table 5.2: Data structures and memory consumption (in MB): join query.

Tuple Format	Table Size					
	100	1k	10k	100k	1M	10M
Map	0.2	1.1	9.3	92.1	920.0	9,131.2
Array	0.2	0.4	2.8	27.3	268.4	2,654.8
POJO	0.2	0.3	1.4	13.6	130.6	1,275.3
Col-Store	0.2	0.3	1.4	13.2	126.8	1,237.0
Col-Store Proj.	0.2	0.3	1.3	11.7	111.5	1,083.8

Discussion: Aggregation Query

A couple of facts in the aggregation tests results are worth mentioning. First, using optimized data structures allowed the aggregation query to operate over windows of larger sizes, which otherwise would not be possible if employing the original non-optimized tuple representations (e.g., it was not possible to run the aggregation query over a window of 100M events when they were represented as Maps or arrays of Objects because these implementations required more memory space than it was physically available).

Second, the POJO format, although more efficient than Map and array representations, suffered severe drops in performance in two distinct points of the graph: from 10k to 100k and from 1M on. The collected metrics reveal that the first drop was primarily caused by microarchitectural aspects (*more specifically, an increase in L2 cache misses*), while the second was due to an increased garbage collection activity (*the percentage of time spent on GC jumped from 11% at 1M to 53% at 10M, and to 83% at 100M*).

The results also indicate that the column-oriented storage model addressed partially or even totally the aforementioned issues. For instance, in contrast with the *POJO* format, the *Col-Store* organization did not suffer with garbage collection issues, resulting in no significant performance loss for large window sizes (i.e., 10M and 100M tuples; see Figure 5.8 (a)). Further, when keeping in memory only the attribute referenced by the query, the column-oriented model was also able to eliminate the microarchitectural issues. This was possible because for that particular implementation (i.e., *Col-Store Proj*.) the memory access pattern is essentially sequential – consuming events from the stream and inserting them into the window means sequentially traversing a primitive type array – which maximizes performance at the microarchitectural level and ensures a steady throughput over the most different window sizes. Indeed, this observation was corroborated experimentally, with the CPI metric remaining basically unaffected in all

tests of that specific implementation (ranged from 0.850 to 0.878 for windows of 1,000 up to 100-million events).

Discussion: Join Query

Interestingly, for the join query the column-oriented storage model did not provide considerable performance gains with respect to the POJO representation[11]. This behavior can be attributed to the fact that the amount of memory consumed by the payload itself (i.e., tuples attributes) in this case is small compared to the overhead of inherent factors of the Java programming environment, such as object alignment, wrappers for primitive types, and especially the heavyweight nature of the `HashMap` class, the structure used for indexing the table in our tests. Since most of the space is consumed by the index, using a more concise representation for the tuples results in very little performance gains. Overall, the final performance of column-store implementation for the join query oscillated around +3% and -5% in comparison with the tests with POJO tuples.

5.5 Improving Algorithms Efficiency at the CPU Level

Nearly all results presented in previous section revealed that the throughput of both aggregation and join queries dropped significantly as their state size increased, even though the employed algorithms have a constant theoretical runtime complexity. Similarly to the tests with Esper, the throughput decrease at the application-level was strongly linked to performance degradation at the micro-architectural level. As shown in Figure 5.9, this correlation was observed not only when using conventional tuple

[11] For the sake of clarity, the lines "*Col-Store*" and "*Col-Store Proj*" were omitted in Figure 5.8 (b) precisely because the results with these two implementations were very similar to the POJO case.

representation formats (i.e., Maps and Objects) but also for the column-oriented organization scheme. In this section we delve into the causes for this behavior and propose optimizations to improve queries scalability with respect to input size.

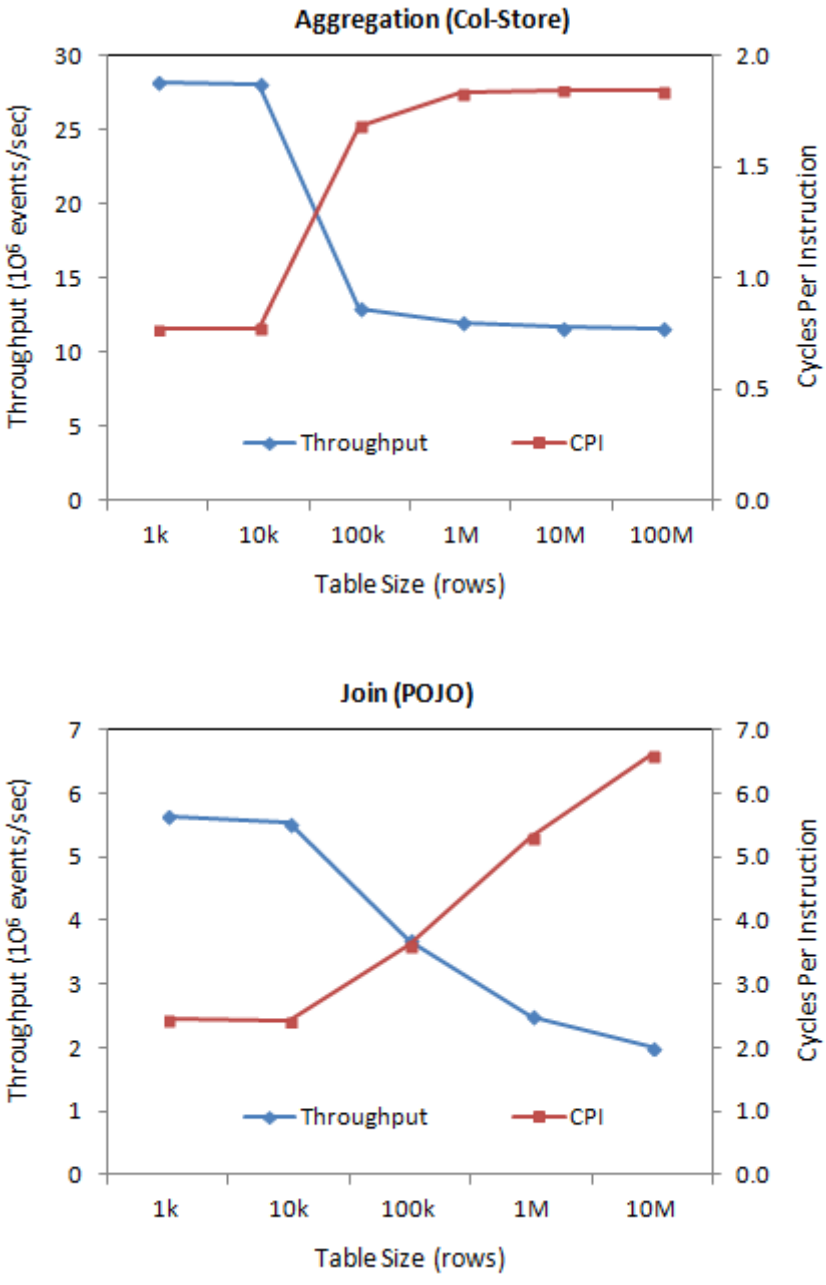


Figure 5.9: Query throughput vs. cycles per instruction.

5.5.1 Aggregation Query

As mentioned before, the algorithm for computing *subtractable* [42] aggregations such as `AVG`, `SUM` or `COUNT` over sliding windows consists essentially in updating some fixed set of state variables upon event arrival while maintaining the events of the window in main memory. As such, the algorithm has a O(N) space complexity but a theoretical O(1) time complexity. In practice, however, several factors can make the running time of the algorithm grow when the input size is increased. One of them is garbage collection: generally, the bigger the working set size and the higher the throughput, the more time will be spent on GCs. Besides, execution efficiency at the CPU tends to be hurt when more elements are referenced due to an increased probability of cache misses. This is particularly the case when events are represented as Objects, because there is no guarantee that consecutive elements in the window will be allocated contiguously in the heap by the JVM. Therefore, even though the algorithm logically traverses the window in a sequential way, the memory access pattern tends to be essentially random. One possible way of eliminating this undesirable effect is to employ the column-oriented storage model, which avoids the random walks through the heap by keeping attributes as arrays of primitive types. However, care should be taken if the number of attributes referenced in the query is large, as in this case, consuming an event from the stream and inserting it into the window will involve accessing several memory locations (one entry per attribute in different arrays).

In order to overcome this issue, we propose a tuned algorithm that minimizes the performance penalty due to multiple inserts. The idea is to avoid references to distant memory locations by using a L2-resident temporary buffer for accommodating the incoming events. This temporary buffer consists in N aligned arrays (one per attribute) as in the original window, but with a capacity of only 100 events. Once these small arrays get full, the events are copied back to the window, one attribute at a time, so that they can be expired later. The algorithm is described in detail in Table 5.3. Figure 5.10

compares the performance of the proposed algorithm with the original column-oriented implementation.

Table 5.3: Cache-aware algorithm for computing sliding-window aggregations.

Input: S: incoming event stream
K: the size of the L2-resident temporary buffer
Output: R: stream of results

for each event *E* in *S* **do**
  for each attribute A_i of event *E* **do**
    store A_i on the corresponding temporary location T_i
  compute aggregation (update aggregator state)
  insert aggregation result into output stream *R*
  if temporary buffer *T* is full **then**
    for each attribute A_i of event *E* **do**
      for each item I_j in temporary buffer T_i **do**
        copy I_j to the appropriate location in corresponding window W_i
    reset the temporary location *T*
    slide the window *W* in K positions

The optimized algorithm provided gains in performance that ranged from 28 to 35 percent when compared to the original column-store. The hardware metrics confirmed that it indeed exploits better the characteristics of the CPU: the CPI was almost half of the CPI of the original column-store and L2 cache miss rate was reduced to around 70% of what was originally measured. Evidently, this microarchitecture-aware algorithm is best-suited for medium-to-large windows, since for smaller sizes the working set of the original column-oriented implementation already fits in L2 cache.

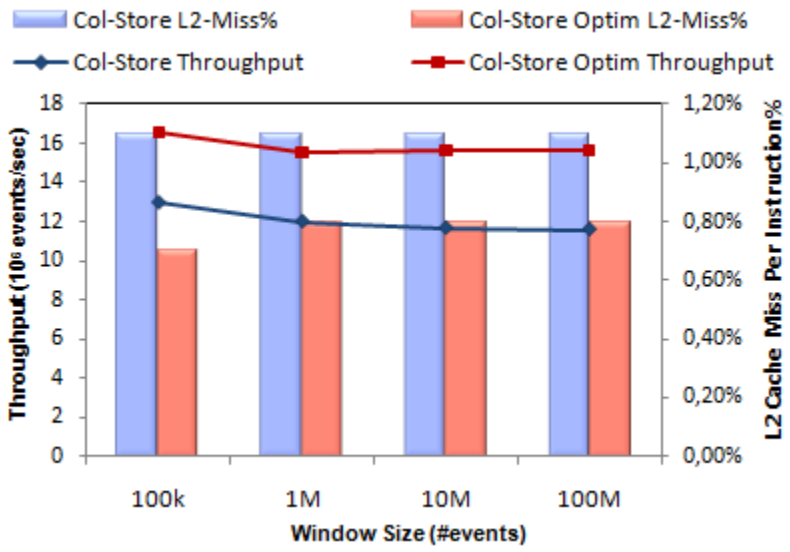


Figure 5.10: Conventional column-store algorithm vs. cache-aware algorithm.

5.5.2 Join Query

In theory, the cost of a lookup on a table indexed with a hash index should be independent on the number of elements stored on it. We have seen, however, that in practice CPU operation – and in particular cache behavior – is considerably affected by working set size and performance tends to drop when larger tables are used (see Figure 5.9). Indeed, further analysis of the hardware metrics collected during the tests of section 5.4 confirmed that the increase in CPI was due to less efficient memory access patterns. For example, as the table size was increased from 1,000 rows to 10-million rows, *L2 cache miss per instruction* metric went from 0.6% to 3.1% and *TLB miss penalty* metric jumped from 0.3% to 19.4%. Notice also that up to 10k rows, the table fits in the 12MB L2 cache, which explains the negligible performance drop from 1k to 10k, and the significant degradation from that point on.

To improve data locality of the join query, we implemented an adapted version of the grace hash join algorithm [55] used in conventional DBMSs. The idea is to reduce the number of times data is brought from main memory to cache by splitting the whole table into partitions and accessing them in bulks. The algorithm works as follows:

- When the table is being populated, the records are stored into partitions using a given partitioning function g (in our tests the table was split into 1,000 partitions);
- Incoming events are then processed in batches. They are buffered into a partitioned list until the batch is complete. (The partitioning function is the same as the one used for splitting the table, which ensures that matching tuples in the batch and the table will be kept in corresponding partitions).
- Once the batch is complete, the corresponding partitions from event batch and table are loaded in pairs. The event partitions are then sequentially scanned, performing for every event a lookup on the corresponding table partition.

Figure 5.11 shows test results with both the conventional hash join algorithm and the batch grace hash algorithm, for table sizes ranging from 10-million to 80-million rows. As it can be seen in the second graph, the batch algorithm indeed improved locality of data accesses, which in turn caused a reduction in average CPI. This resulted in performance gains that ranged from 11% to 44%, as illustrated in the uppermost graph.

Notice that there are a couple of competing factors influencing the performance of the batch grace hash algorithm. For example, each table partition should ideally fit in L2 cache in order to minimize the high penalties associated with memory accesses. Assuming that table size is application-specific and cannot be changed, the only way this can be achieved is by increasing the number of partitions in the table. Doing so, however, means that the number of partitions in the batch of events is also increased, thus reducing the number of events per partition. A reduced number of events per partition will probably hurt performance as a good fraction of the lookups in the table will incur in compulsive cache misses. For avoiding this to happen, batch size could be increased in the same proportion as the number of partitions, but obviously this is only feasible if there is availability of memory resources.

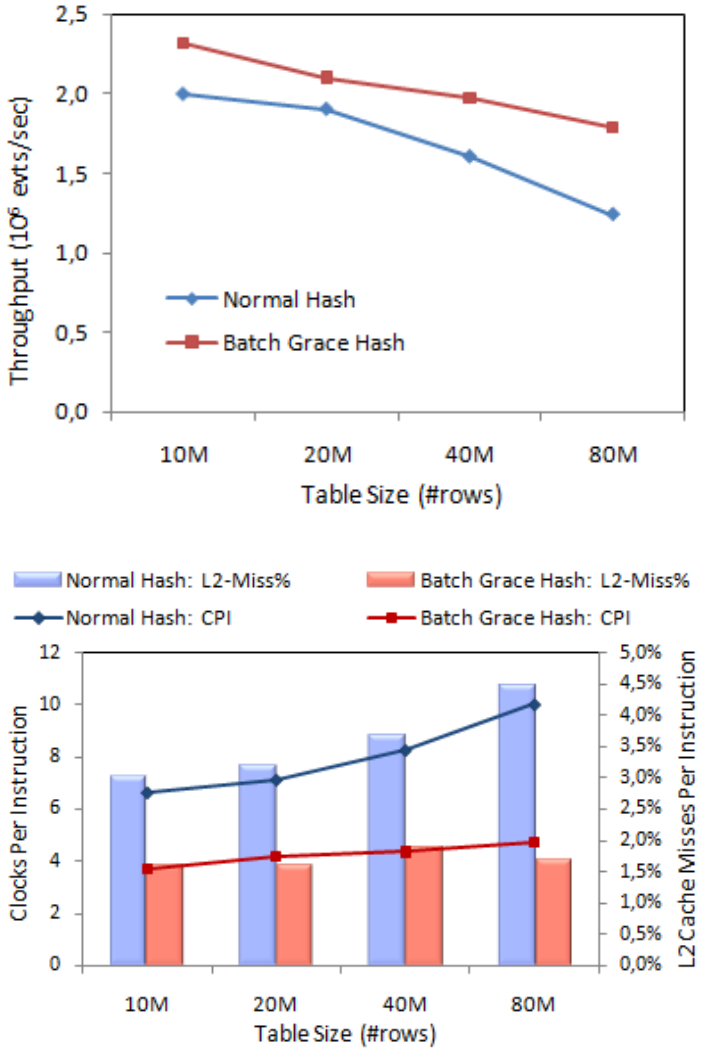


Figure 5.11: Conventional hash join vs. batch grace hash join.

5.6 Multi-Query Scenario

A natural question that might arise after analyzing the results of previous sections is whether similar improvements in performance would be observed once we move from a scenario with only one continuous query running at a time to a multi-query scenario. In this section we answer this question and present the results for a set of tests in which the proposed optimizations are validated by measuring system performance during the

execution of multiple simultaneous queries. More specifically, we tested three different situations:

i. N instances of the same query are computed over a *single event stream* in a *single thread*;

ii. N instances of the same query are computed over *independent but identical event streams* in a *single thread*;

iii. N instances of the same query are computed over *independent but identical event streams* in *N threads*.

For aggregation, we performed tests with 1 up to 16 simultaneous queries, with sliding windows of 10 million events each. For join queries, we tested 1 up to 8 simultaneous queries operating over N tables of 10 million records (there was no physical memory available to test with 16 queries). We then analyzed the evolution of throughput and hardware-level metrics as we progressively added more queries to the configuration. The output throughput of each setting is shown in Figure 5.12.

Application-level and hardware-level metrics collected during tests indicate that the proposed optimizations are also effective in a multiquery scenario. For instance, the microachitecture-aware aggregation algorithm introduced in section 5.5.1 achieved superior performance than the conventional column-store in all multi-query tests. Also, the "*Col-Store Proj*", which achieved the highest throughputs in the single aggregation query scenario due to improved microarchitectural performance, was once more the best performing implementation.

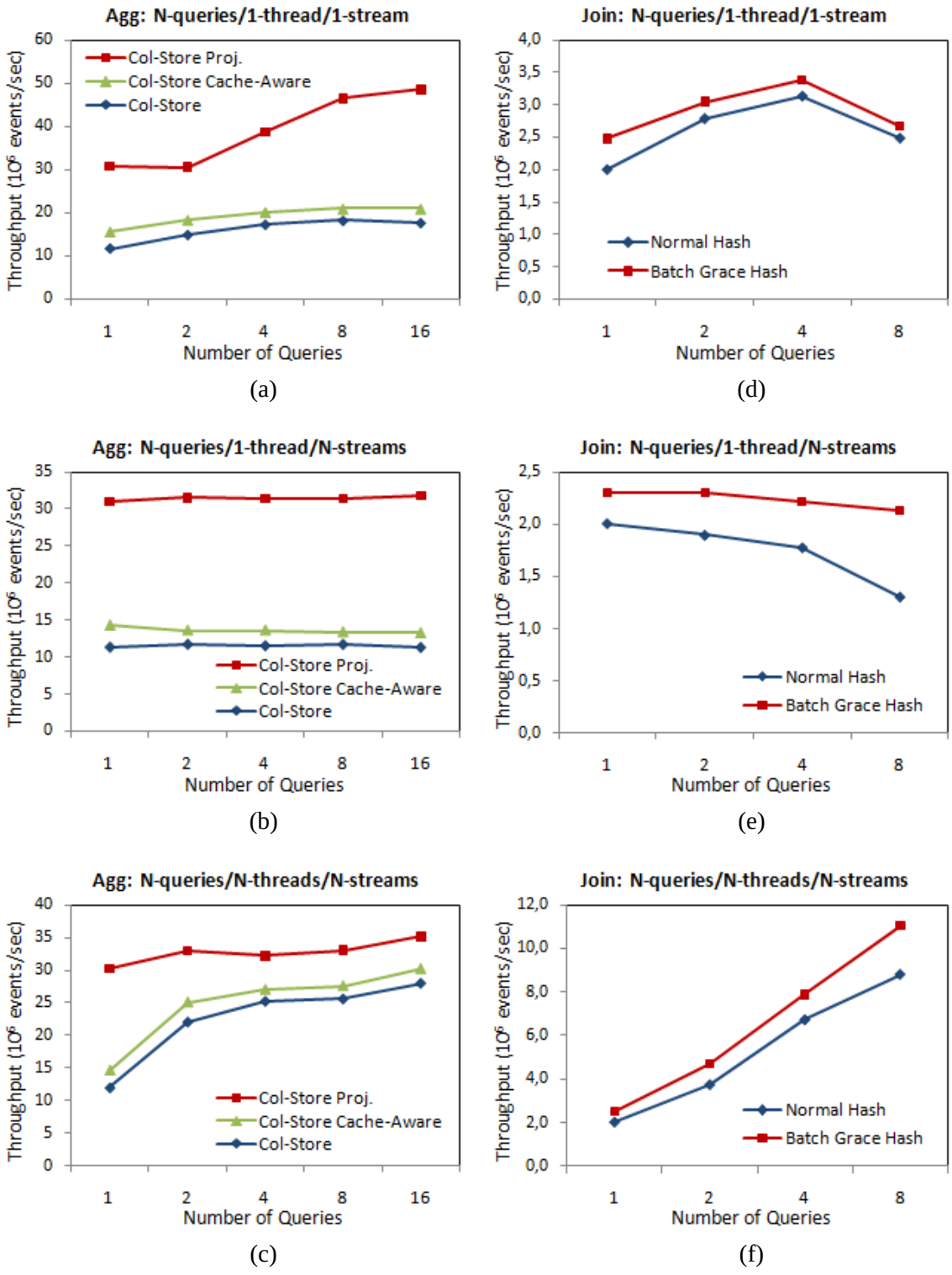


Figure 5.12: Optimizations in a multi-query scenario.

With respect to the join query, the speedup of the batch grace hash algorithm over the conventional hash algorithm oscillated from 1.02 to 1.64. On average, the optimized implementation achieved 22% more throughput than the conventional non-optimized one, a slightly better speedup than the one observed in the single query scenario (20%). Once more, hardware-level metrics followed the trend observed in the single query scenario (on average, the conventional hash algorithm had a CPI of 7.52 and L2 cache miss rate of 3.5% against a CPI of 3.46 and a L2 cache miss rate of 1.6% for the batch grace hash algorithm).

A few words about the shape of the curves: on (a) the output throughput increased sligthly when more queries were added as a result of a reduction on the relative weight of event instantiation on the workload (event is created once but processed N times). This constrasts with the workload on (b) where event instantiation is replicated in the same proportion as queries, which explains the steady line (individual throughput of queries decreases, but total system throughput remains the same). Interestingly, the curves on (c) did not present a linear (*or close to linear*) growth that one would expect when increasing the amount of resources for a set of independent tasks (as happened on (f), for example). We found out that this happened because the CPI metric in these tests increased essentially in the same proportion as the number of queries. The reason for this behavior, however, is unclear to us since the other hardware metrics (i.e., cache miss rates, instruction fetch stalls, resource stalls, etc.) did not show any change that could justify this increase in CPI. Finally, it should be noticed that the drop on performance when jumping from 4 to 8 queries on (d) was caused by increased garbage collection activity (for 8 queries the system ran close to the maximum memory available).

5.7 Optimizations on the EP System

We now examine how the proposed optimizations affect the performance of the EP engine Esper. Figure 5.13 below shows the maximum throughput and memory consumption for the aggregation query, using different representation formats.

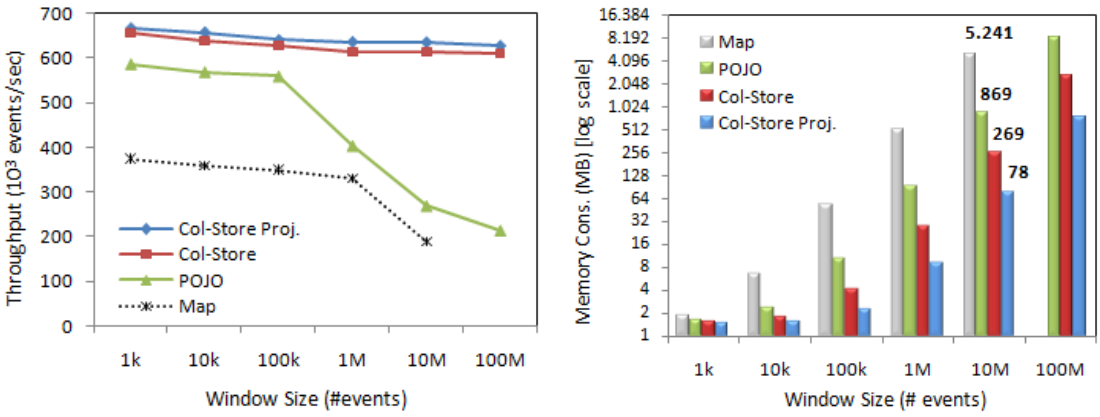


Figure 5.13: Optimizations on Esper (aggregation query).

As in the tests with the EP prototype, the modified version of Esper using a column-oriented storage model achieved higher throughputs and scaled better than the original implementations using Maps or POJOs as event representation. The column-oriented implementations also proved once more to be useful for reducing memory consumption for aggregation queries. Interestingly, the throughput difference between the "*Col-Store*" and "*Col-Store Proj.*" implementations was significantly smaller in the tests with the Esper engine (around 2.7%) than the one observed with the prototype (2-fold increase). This is probably related to the fact that the maximum throughputs achieved by Esper were considerably lower than the obtained by the prototype. Thus, with other bottlenecks limiting maximum achievable throughput, the net effect of the optimizations is reduced. For the same reason, the optimizations specifically targeted at the microarchitectural level had only modest effects on Esper: both the cache-aware aggregation algorithm and the batch hash grace join algorithm resulted in an improvement of around 1 percent – situating them in-between the "*Col-Store*" and

"*Col-Store Proj*." results, like in the tests with the prototype. These results indicate that the relative weight of the aggregation operation itself is small compared to the overall processing path of tuples on Esper. Thus, higher gains should be observed in scenarios with multiple queries (where their relative weight is higher) or if other bottlenecks of the engine are minimized or removed.

5.8 Summary

In this chapter we investigated ways to improve the efficiency of event processing systems. In particular, we demonstrated that the excessive memory consumption observed in our previous study is related to a bad choice of data structures used for maintenance of events payload. We also verified that microarchitectural aspects play a fundamental role on the performance degradation observed when query state increases. We then proposed, implemented, and evaluated changes in data organization and novel algorithms to improve resource utilization and execution speed of continuous queries. Specifically, we tested a column-oriented approach, where attributes of incoming events are stored independently, in aligned arrays of primitive types, rather than as regular Objects. This alternative organization scheme resulted in significant memory savings (up to 67 times less space required) and, in some cases, in considerable gains on query throughput (e.g., a 35-fold increase, for aggregation queries). We also proposed cache-aware algorithms for aggregation and join operations in order to minimize the performance penalty suffered by queries with large state sizes. Experimental results revealed an improvement of up to 35% for aggregation queries and 44% for joins.

Chapter 6

Performance Enhancements for EP Systems - Part II : When Memory is not Enough

In the last chapter we proposed algorithms and evaluated different data organization schemes focused in improving the execution of continuous queries at the CPU-RAM level. In this chapter we move our focus to large-scale event-driven applications, whose state does not fit in main memory. We introduce *SlideM*, an optimal buffer management algorithm that handles memory shortages by sending portions of large sliding windows to disk. We also extend the proposed algorithm and devise a strategy to share computational resources when processing multiple queries over overlapping sliding windows. We implement both techniques in a real event processing platform and demonstrate that they scale for input rates as high as 300,000 events per second, while consuming small amounts of memory and putting minimal load on the I/O subsystem.

6.1 Introduction

Many event-based applications involve the computation of *aggregates over sliding windows*, which allow users to better measure the quality-level of their businesses and systems in real-time. For example, consider a call-center monitoring application where

information about customers' calls is constantly analyzed by an event processing system. A typical query executing at the EP engine in such scenario is:

Query 1: "*Report, every second, the average time during which customers are waiting for service, across the last hour.*"

```
SELECT AVG(waitTime)
FROM   calls [RANGE 1 HOUR SLIDE 1 SECOND]
```

The query above, expressed using the CQL [9] language syntax, specifies an AVG aggregation query over a sliding window with two parameters: RANGE, which defines the span of the window (*i.e., for how long tuples of the event stream "calls" are considered in query answer computation*); and SLIDE, which defines when tuples are expired out of the window and controls the frequency in which updated results are produced.

As discussed in section 2.1, EP systems use main memory for processing their continuous queries. Unfortunately, many queries have an unbounded space cost, which cannot be determined beforehand. For instance, the memory consumption of Query **1** typically depends on the event arrival rate, which might vary significantly during query execution. In those circumstances, one would expect EP systems to be prepared to handle memory shortages. However, we have demonstrated in Chapter 4 that many commercial EP engines deal badly with this situation – some suffer from thrashing due to OS paging or excessive garbage collection activity while others simply crash with out-of-memory errors.

In this chapter we address this problem for analytic workloads composed by a large number of continuous aggregation queries. We introduce *SlideM*, a buffer management algorithm that selectively offloads sliding windows state to secondary storage when main memory becomes insufficient. Our approach is based on the observation that for most aggregation queries the memory consumption and access pattern is dictated by the

sliding window, and that this access pattern can be exploited to achieve excellent performance while using small amounts of main memory.

This work was in part motivated by the scalability problems faced by real event-based applications we had contact with, while working in cooperation with industrial partners. Most of these applications required computing several metrics (aggregates) over very large sliding windows, under stringent memory requirements. Throughout the rest of the chapter, we use one of such use-cases, the call-center monitoring application introduced earlier and described in detail in section 6.5.2, to illustrate the problem. The requirements of this application include the computation of about 150 KPIs over 24-hour sliding windows, under an input rate of around 1,000 events per second, running on a machine with 2 gigabytes of main memory. Note that, given the moderate input rate, relative simplicity of the KPIs being computed (SUM, COUNT and AVG aggregates) and the large size of the window, the application tends to be memory-bound rather than CPU-bound. A simplistic calculation gives an idea on the dimension of the problem: considering that the average event size is 94 bytes, and that all tuples need to be maintained until they are expired out of the window, a single aggregation query will require at least: 1000 X 24 X 3600 X 94 bytes = 7.6 gigabytes (*we show in Section 6.2 that the commonly used technique of pre-aggregating data instead of storing tuples may require even more space*). Our goal is to allow such memory-intensive applications to run smoothly on an EP engine whether they fit on available memory or not.

6.1.1 Summary of Contributions

In this chapter we make the following contributions:

- We analyze current proposals for executing sliding-window aggregates and show that frequently-used techniques, designed to reduce memory consumption and create opportunity for resource sharing, in many cases do not produce the desired effects (Section 6.2).

- We propose *SlideM*, an optimal buffer management algorithm to deal with memory shortages during execution of sliding-window aggregation queries. We demonstrate that, contrary to common sense, storing windows data on disk can be appropriate even for applications with very high event arrival rates (Section 6.3).
- We build upon the proposed buffer management algorithm and develop a strategy (*SSM*) to share computational resources when processing multiple aggregation queries over overlapping sliding windows (Section 6.4).
- We implement our proposed techniques in a real EP system [75] and validate their effectiveness through an extensive experimental evaluation (Section 6.5).

6.2 Background: Sliding-Window Aggregates (SWA)

Continuous queries in EP engines are computed over infinite event streams rather than bounded datasets. However, many operations cannot be executed over infinite inputs in bounded memory, and some *blocking* operations require seeing the entire input before producing any result (e.g., join) [40]. Traditionally, these two issues have been addressed by limiting the amount of data over which operations take place through the use of *sliding windows*. As discussed in section 4.2, a sliding window is a construct that retains only the most-recently arrived tuples of an event stream. The *size* of a sliding window determines the amount of data to be retained, and can be specified in number of tuples (*count-based* windows) or through an interval (*time-based* windows). Stale tuples are purged when the window *slides*, due to arrival of new event or time passing.

A *sliding-window aggregate* (SWA) computes an aggregation function over a sliding window content and produces an updated result every time the window slides. For example, consider our motivating scenario where a stream of statistics continuously generates new information about call-center interactions with its customers. Query 1

defines a sliding window that retains the data items that arrived in the last hour, computes the average "waiting time" from this set of elements, and reports a new result every second. A common variation of this query structure is to have a *grouped aggregation*, where the input stream is logically partitioned into sub-streams, based on a grouping key, and one aggregate is produced for each partition. For instance, a GROUP-BY clause could be added to Query 1 in order to produce the average waiting time per customer region or per employee.

Sliding-window aggregates are recognized as one of the fundamental operations of EP platforms and have been extensively studied in previous work [8] [12] [56] [59]. In the rest of this section we discuss how SWAs have been traditionally implemented. We present the two most frequently-used approaches, and compare how well they utilize memory resources in different workload scenarios.

6.2.1 SWA Implementation

Many important aggregates such as AVG, SUM, COUNT, MIN and MAX can be computed incrementally, in a single-pass over data items. This, in principle, allows an aggregation operator to discard events right after they have been processed. For instance, an AVG aggregate can be computed in O(1) space by simply keeping two variables – *sum* and *count* – and updating them upon event arrivals. However, when the aggregate is applied over a sliding window, tuples are eventually expired and this tuple removal has to be reflected into the query answer – in the AVG example, this means subtracting from the *sum* variable the value of the aggregated field in the expired tuples and decrementing the *count* variable by the number of expired tuples. Therefore, an SWA operator needs to maintain information about events that arrived previously so that the result can be updated properly when they eventually leave the window.

Traditionally, two approaches have been used to keep this information about past events in an SWA. The first, simply keeps all tuples in the window until it is time to expire them [9]. Normally, the sliding window is an operator by itself, which forwards

incoming and expired tuples to subsequent, aggregate operators (Σ), as depicted in Figure 6.1.

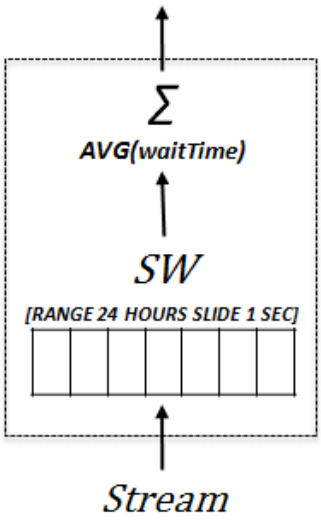


Figure 6.1: Query plan of a SWA using the single-window (1W) scheme.

The second approach, first introduced in [59], and adopted in subsequent proposals (e.g., [56]) sub-aggregates the incoming stream using smaller windows and then aggregates these sub-aggregates into a window of the original size in order to produce the final query result (*see Figure 6.2*). Taking Query **1** as example, SUM and COUNT sub-aggregates are computed over a 1-second window and then aggregated over a 1-hour window, thus producing the final result. The main advantage of this *two-level aggregation* (2LA) scheme over the former *one-window* (1W) approach is that the space cost of the query no longer depends on the input rate. However, as we are going to discuss next, it does not guarantee a bounded space cost, and for some workloads, it actually results in increased memory consumption.

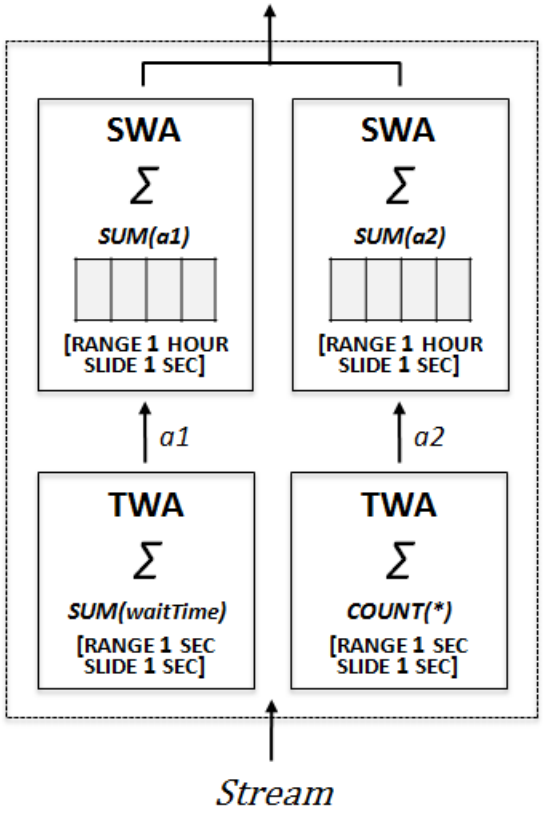


Figure 6.2: Execution plan for SWA Query 1 using the 2LA scheme.

6.2.2 Space Cost Analysis

In this section we examine the space cost of the two widely-used SWA implementations. In particular, we demonstrate that either approach can incur in considerable memory costs, eventually bringing event processing applications to run out of memory. We also show that the *2LA* technique, originally designed to reduce space cost of SWAs, might in many cases aggravate the problem.

Two-Level Aggregation (2LA)

The *2LA* technique can be very useful for reducing space and computation cost of periodic sliding-window aggregates, particularly when input rates are high and/or the answer does not need to be updated often. However, there are a couple of issues that

limit its effectiveness in many important scenarios. Probably the most relevant of them is that its space cost grows linearly with the number of aggregates being computed. This is particularly critical since most monitoring applications compute not one, but several aggregates – either because different aggregation functions are needed, distinct sets of attributes of the input streams are aggregated, or different grouping criteria are used. For instance, the users of the call-center monitoring application are interested not only in the *average waiting time*, but also in the *total* waiting time, the average *call time*, and the average waiting time *per region*. With the *2LA* scheme, each such aggregate results in a pair of operators, a *tumbling-window aggregate* (TWA) and a subsequent *sliding window aggregate* (SWA), as illustrated in Figure 6.2.

Another issue is that aggregations with a GROUP-BY clause implemented using the *2LA* scheme have their space and computation cost directly affected by the number of groups. This is because the TWA operator will produce as much aggregates as the number of distinct groups seen during the lifetime of its window. Each of these aggregates consumes space and computation in the subsequent SWA operator. Moreover, since the number of groups seen during the interval of the TWA window cannot be determined a priori, the *2LA* scheme does not guarantee a bounded space cost for grouped sliding-window aggregates. Taking into account the aforementioned factors, the total space cost when computing a set of aggregates using the *2LA* scheme is determined as follows[12].

Consider that N sliding-window aggregates, $\Sigma_1,...,\Sigma_N$, are to be computed – *each representing a unique combination (Φ_i, F_i, P_i) of aggregation function (Φ), aggregated fields (F) and grouping criteria (P)* – over a common sliding window with size W and

[12] For the sake of brevity, we limit our discussion to time-based windows, but similar analysis applies to count-based windows.

update interval U. The space cost of each aggregate Σ_i is given by the sum of the costs of its inner tumbling-window and sliding-window aggregates:

$$Space_{2LA} = N \cdot (Space_{TWA} + Space_{SWA})$$

The TWA operator does not keep tuples in a window and only consumes the space required to maintain an aggregation state s' for each of the g groups seen during period U as shown below:

$$Space_{TWA} = g \cdot s'$$

The SWA operator, on the other hand, keeps both the tuples produced by TWA and a per-group aggregation state:

$$Space_{SWA} = \frac{g}{U} \cdot W \cdot t_{agg} + G \cdot s$$

In the formula above, $\frac{g}{U}$ is the rate at which tuples arrive at SWA from TWA, t_{agg} is the size of the tuples produced by TWA, G is the total number of groups seen during W, and s is the size of the aggregation state per-group. The final space cost of the *2LA* scheme for A aggregates is then given by the formula below:

$$Space_{2LA} = N \cdot \left[g \cdot \left(s' + \frac{W \cdot t_{agg}}{U} \right) + G \cdot s \right] \quad (6.1)$$

Note that depending on the function being computed, the aggregation state sizes s and s' can be constant or grow with the number of tuples in the corresponding window. As discussed elsewhere ([8] and [59]), *subtractable* aggregates like SUM, COUNT, AVG, and VARIANCE can be computed with constant storage, but *distributive* (e.g., MIN and MAX) and *holistic* (e.g., QUANTILE) functions require O(N) space.

To Sub-Aggregate or not to Sub-Aggregate

We now examine the space cost of the *one-window* approach. As it can be seen from Figure 6.1, the memory consumption for the *1W* scheme corresponds to the sum of the

space required to maintain the tuples in the main sliding window, and the state of each aggregate Σ_i. The former is obtained by the product of the window size *W*, the input rate λ, and the tuple size *t*. The latter is given by the product of the number of aggregates N, the total number of groups *G* seen during *W*, and the state size of each aggregate operator *s*. Or algebraically:

$$Space_{1W} = \lambda \cdot W \cdot t + N \cdot G \cdot s \quad (6.2)$$

We can see from formulas (6.1) and (6.2) that each approach is more sensitive to a given factor than the other. With the *1W* implementation, the space cost will be substantial for large windows if the input rate is high. On the other hand, the *2LA* scheme is immune to the input rate[13], independently on how large the window is, but can be severely penalized if the workload has many aggregates or groups.

As an example, we compare the memory consumption of the two different approaches using parameters taken from the call-center monitoring use-case. Let W=24 hours, U=10 seconds, λ=1000 events/sec, N=104 aggregates, G=10000 groups, g=1000 groups, s=s'=16 bytes[14], t=94 bytes, and t_{agg}=20bytes. The space cost of each scheme is in this case:

- $Space_{2LA} = 104 \cdot \left[1000 \cdot \left(16 + \frac{86400 \cdot 20}{10}\right) + 10000 \cdot 16\right] \cong \mathbf{17GB}$
- $Space_{1W} = 1000 \cdot 86400 \cdot 94 + 144 \cdot 10000 \cdot 16 \cong \mathbf{8GB}$

As we can see, for this particular use case, performing a *two-level aggregation* in the end results in less efficient usage of memory resources than when computing the aggregates with a single window. More importantly, considering that in this application

[13] Assuming that the number of groups *g* seen during period U is not affected by the input rate, which typically is not the case.

[14] Only *subtractable* aggregates are computed. We simplify discussion using a single value to represent the average state size of the different aggregation functions in the application.

the available memory is limited to less than 2GB, neither of the two approaches allows the workload to run entirely at RAM. In this situation, it is necessary to selectively spill part of the queries state to disk so that the application does not run out of memory. Note that in either scheme the queries space cost is largely dominated by the state of the sliding window(s). For this reason, we address the problem of insufficient memory resources during computation of SWAs with an algorithm to manage the content of sliding windows.

6.3 The *SlideM* Buffer Management Algorithm

In this section we introduce *SlideM*, an algorithm for managing the working set of sliding windows. The proposed algorithm exploits the fact that sliding-window operators are most of the time manipulating only a small fraction of their data set and are doing so in a very predictable pattern – once a tuple is stored on the window it is not going to be accessed by the sliding-window operator until it is time to expire it, which may take long (e.g., consider a 6-hour time-based window).

SlideM is employed on a per-operator basis, that is, each window physical operator in the query plan is given a *repository* to hold its tuples. The actual location of tuples (either RAM or disk) is encapsulated by this repository, which internally implements a buffer management strategy based on *SlideM*. The repository includes a buffer pool (BP) for holding the memory-resident part of the window and a handle for accessing tuples at secondary media. Both the buffer pool and the data file at disk are divided in non-spanned blocks with a fixed block factor. These blocks are the unit of transfer between main memory and disks.

The algorithm operates as illustrated in Figure 6.3: when the buffer pool gets full, it first sends to disk the block containing the most recently arrived tuples because these are the ones that are not going to be needed for the longest time. Similarly, when the oldest block at RAM is expired and hence the BP has space left once more, *SlideM* brings

back from disk the least recently written (LRW) block, because it contains the tuples which are going to be needed next among the ones currently at disk. This behavior ensures that memory will always contains the tuples that are going to be needed by the sliding-window operator in the shortest time. The algorithm operation is described in details in Figure 6.4 and Procedures 1 and 2.

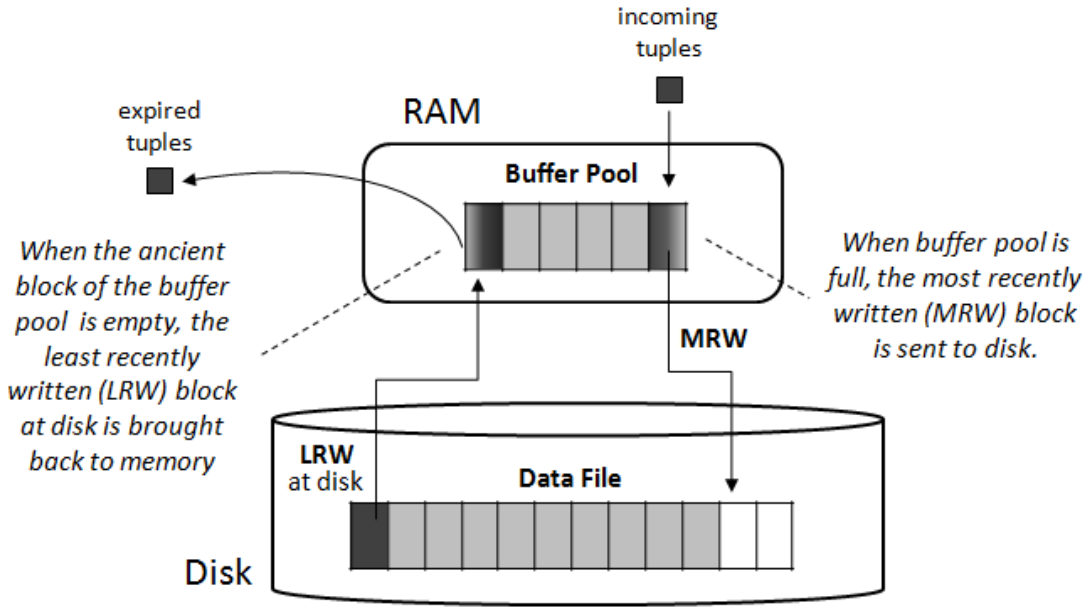


Figure 6.3: Overview of SlideM operation.

Procedure 1 describes event arrivals: every time a new tuple needs to be stored in the window, the algorithm checks whether the block at the tail of the window still has space left. If so, it stores the tuple normally at the end of the block; otherwise it allocates a new block at the buffer pool (*as shown in step 2 of Figure 6.4*). If the buffer pool is full (*3*), the algorithm first spills the most recently written (MRW) block to disk (*4*) to free space for the new block that will be soon allocated. The MRW block is written at a free position on disk (*9*) or at the end of the data file, if there are no free blocks (*5*).

Procedure 1 add(Tuple t)

let *recent_block*: block at RAM holding recently arrived tuples

```
1:   if recent_block is full then
2:     if buffer pool is full then
3:       if there are free blocks at disk then
4:         diskPos ← address of least recently released block
5:       else
6:         diskPos ← end_of_file
7:       end if
8:       WRITETODISK(diskPos, recent_block);
9:     end if
10:    recent_block ← ALLOCATENEWBLOCK();
11:  end if
12:  recent_block.APPENDTUPLE(t);
```

Procedure 2 expireOldest()

let *ancient_block*: block at RAM holding soon-to-expire tuples

```
1:   if ancient_block is empty then
2:     buffer_Pool.REMOVE(ancient_block);
3:     if there is data at disk then
         // position at disk of least recently written block.
4:       lrw ← GETLRWDISKBLOCKADDRESS();
5:       lrwDiskBlock ← READFROMDISK(lrw);
6:       buffer_pool.ADD(lrwDiskBlock);
7:     end if
8:     ancient_block ← buffer_pool.GETLRWBLOCK();
9:   end if
10:  ancient_block.REMOVEOLDESTTUPLE();
```

Tuple expiration happens as in Procedure 2: when the repository receives a request to remove a tuple at the beginning of the window it checks whether the oldest block is now empty. If so, the block is removed from the buffer pool (*as shown in step 6 of Figure*

6.4); if there is data at disk, the LRW disk block is brought to the buffer pool, at the position of the just-dismissed block (*7*). Then, the tuple is finally removed from the (newly arrived to memory) ancient block.

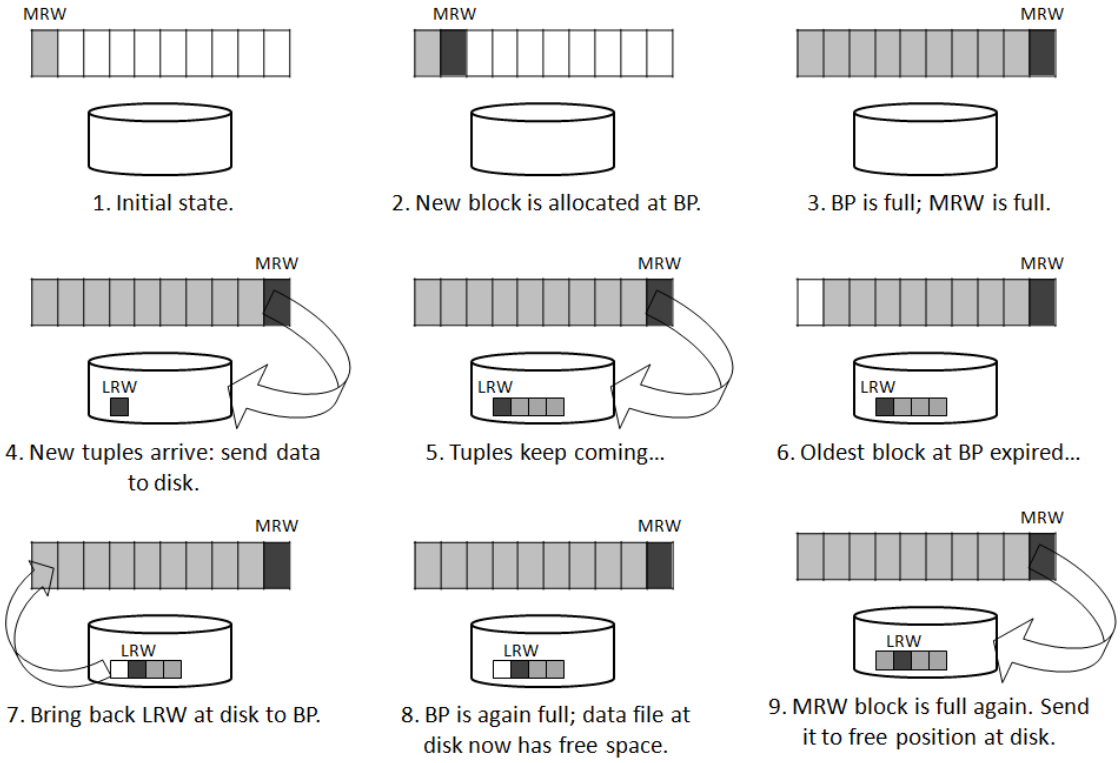


Figure 6.4: *SlideM* algorithm in the several phases of its execution.

It should be clear that *SlideM* operation results in a very small memory consumption. In its strict sense, the algorithm needs only the equivalent to two blocks, one for holding the oldest part of the window (*ancient block*) and another for accommodating the newly arriving events (*recent block*). In fact, the performance of *SlideM* should not be affected by buffer pool size, unless BP is large enough to hold the entire working set of the window – if the window does not fit at RAM the algorithm will necessarily be swapping data to/from disk. More importantly, *SlideM* is optimal in terms of the amount of generated I/O as it always evicts to disk the block that is not going to be referenced

for the longest time (*the optimality of clairvoyant page replacement policies has been first established at [14]; a short proof can be found at [78]*).

6.3.1 Discussion: I/O Load

We now examine the I/O demand of the *SlideM* buffer management algorithm. As explained before, *SlideM* issues disk *read* requests every time blocks at RAM are expired and performs disk *write* operations whenever a new block needs to be created at buffer pool but there is no space left. Thus, the number of I/O operations requested per second (IOPS) by *SlideM* for a single sliding window operator is given by:

$$IOPS = Expired\_Blocks/sec + New\_Blocks/sec$$

Assuming that events are fixed-size and there is a balance between event arrival and expiration rates – which is always true for count-based sliding windows and is also frequently the case across a period [τ, τ+WINDOW_RANGE] of a time-based sliding window – the following property holds:

$$Expired\_Blocks/sec = New\_Blocks/sec$$

From which we derive:

$$IOPS = 2 \cdot New\_Blocks/sec$$

Now, the rate at which new blocks are produced is a function of the event arrival rate, λ, as follows:

$$New\_Blocks/sec = \lambda / block\_factor$$

Where *block_factor* represents the number of tuples stored inside a block. This relation gives us the final I/O demand:

$$IOPS = 2 \cdot \lambda / \lfloor block\_size / tuple\_size \rfloor \quad (6.3)$$

$$IO_{bandwidth} = IOPS \cdot block\_size \quad (6.4)$$

EXAMPLE: For an input rate of λ=1000 events/sec, 94-bytes tuples, as found in the call-center use-case, and a block size of 64KB, the IO demand of *SlideM* will be (*assuming the 1W scheme is used*):

$$IOPS = 2 \cdot 1000 / \lfloor 64 \cdot 1024 / 94 \rfloor = 2.9\ iops$$
$$IO_{bandwidth} = 2.9 \cdot 64 / 1024 = 0.18\ MB/sec$$

Note that these numbers are far less than the theoretical transfer rate of modern hard drives (e.g., up to 204 MB/sec [81]), or the maximum measured disk bandwidth achieved under workload conditions similar to the modeled application (around 25MB/sec). Therefore, *SlideM* is capable of handling much larger input rates than the ones mentioned so far or to process a much larger number of simultaneous sliding window operators before the I/O subsystem starts to become a bottleneck. Nevertheless, the scalability of the algorithm can still be greatly improved by sharing the content of overlapping windows as we discuss next.

6.4 Sharing State of Overlapping Sliding Windows

The previous section introduced *SlideM*, an efficient algorithm to manage the state of a *single* sliding window operator. Now we extend the discussion to a multi-query scenario, where *multiple overlapping* sliding windows are defined over a common event stream. The problem is of foremost importance as large-scale monitoring applications usually process several aggregation queries over different time granularities – e.g., average price of a stock in the last hour, 12 hours, last day and so on. In a naïve approach, each of these overlapping windows would be mapped into an operator inside the query execution plan. Obviously, this limits system scalability and performance since having one operator per window implies that tuples (*or pointers to tuples*) are stored multiple times at different places, thus wasting memory space. Using the algorithm of the last section only address partially this issue as the bottleneck is eventually moved from the memory system to the I/O subsystem. We then build upon

the *SlideM* algorithm and propose a shared execution scheme we call *Shared SlideM* (*SSM*) to improve the usage of computational resources when processing multiple overlapping sliding windows.

6.4.1 Shared SlideM (SSM)

We consider the problem of processing a set of N aggregation queries over N sliding windows of different sizes, defined over a common event stream S. For example, assume that three SWA queries are defined over a stream "*calls*" as follows:

```
Q1: SELECT AVG(waitTime)
    FROM calls [RANGE 1 HOUR]

Q2: SELECT AVG(waitTime)
    FROM calls [RANGE 6 HOURS]

Q3: SELECT AVG(waitTime)
    FROM calls [RANGE 12 HOURS]
```

A direct translation of this set of queries would result in an execution plan like the one shown in Figure 6.5, with aggregation (Σ) and sliding window (ω) operators being replicated for every query in the set.

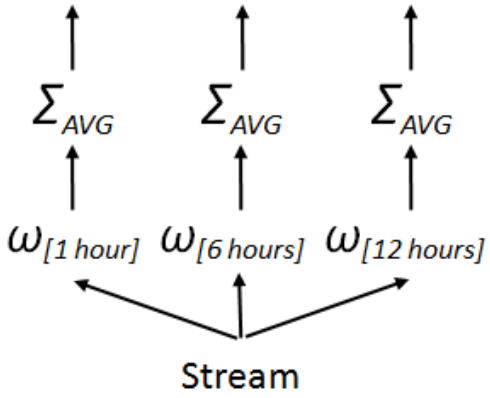


Figure 6.5: Unshared execution plan for three SWA queries.

This naïve approach simplifies query plan generation, but wastes memory during query execution, since the tuples stored in the smaller windows are also, by definition, present in the larger windows, and could be instead maintained in a single, shared, location. This is illustrated in Figure 6.6: the tuples that arrived in the interval [τ_{now} - *1hour*, τ_{now}] are part of all three windows; similarly, tuples belonging to the interval [τ_{now} - *6hours*, τ_{now} - *1hour*] are shared by both the 6-hour and 12-hour windows. In the end, having one operator per window implies that roughly half of the tuples in the query set are stored more than once.

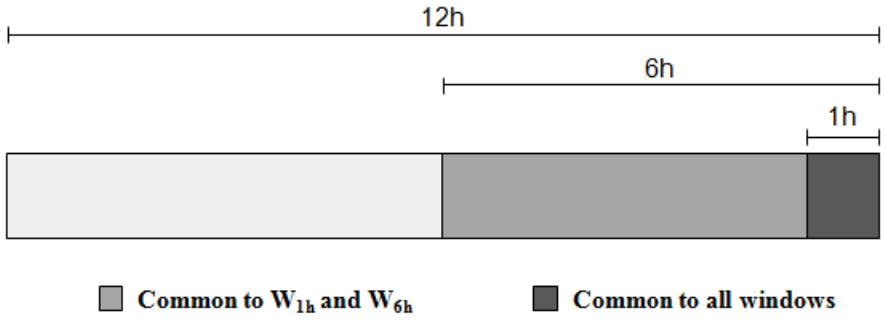


Figure 6.6: Three overlapping sliding windows.

Using the *SlideM* algorithm reduces the pressure over main memory since portions of the windows can be offloaded to disk, but does not solve the problem of unnecessary data redundancy. Moreover, assuming that the windows do not fit in main memory, each query will produce a pair of IO operations from time to time (*read* for the ancient part of the windows and *write* for the recent segment). Eventually, as the number of queries increases, the I/O subsystem will become saturated.

To overcome these issues, we propose *SSM*, an adaptation of the *SlideM* algorithm in which multiple overlapping sliding windows are processed in a shared way. *SSM* works much like *SlideM*, in the sense that it sends parts of the window to disk when main memory is insufficient and brings data back from disk when it is time to expire them. However, contrary to *SlideM*, *SSM* manages a tuple repository that serves multiple *logical* window operators. We use the term ‘logical’ here because the several windows are in fact

implemented by a single operator (Ω) as illustrated in Figure 6.7. This allows sharing computation of tuple arrivals as we explain next.

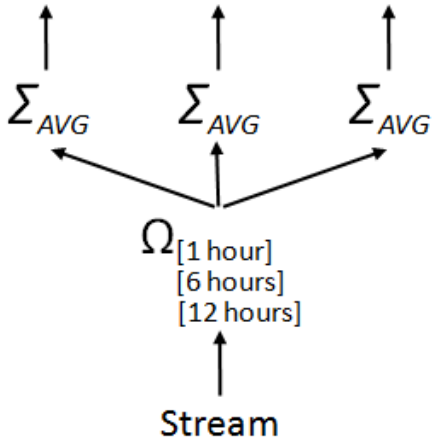


Figure 6.7: Shared execution plan for three SWAs.

A *SSM* tuple repository shared by multiple overlapping windows looks like the structure shown in Figure 6.8. The recent block (MRW), which stores the newly-arriving tuples, is common to all windows, but each window has its own ancient block (LRW), containing the tuples which are about to expire.

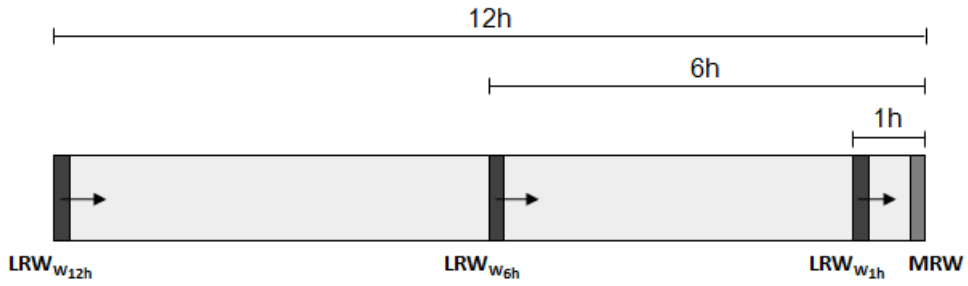


Figure 6.8: Shared *SSM* tuple repository serving multiple windows.

As the recent block is shared by all windows in the set, incoming tuples are processed only once by the shared operator Ω. Tuple arrivals in *SSM* occur essentially in the same way as in *SlideM* (see Procedure 1), with the exception that the request for adding tuples in the repository is now shared by multiple windows and the block sent to disk when the

buffer pool is full is not necessarily the recent block (line 9 in Procedure 1) – a victim block must be selected instead. The major differences are, though, on the way tuple expirations are handled: first, tuples are not purged out of the repository unless the window which requested expiration is the largest one in the set. Intuitively, a tuple can only be discarded when it no longer belongs to any window, which happens when the largest window in the set requests its expiration – the same holds in a, coarser, block-level granularity. Another difference is that *SSM* does not prefetch data from disk when a block at RAM gets empty as *SlideM* does. This is because the LRW block at disk is not necessarily the block which is going to be needed next by the set of windows – with multiple windows the disk access pattern is no longer strictly sequential. Since determining which block will be required next is a potentially expensive operation, *SSM* skips pre-fetching and only brings data from disk when a request to a non-memory-resident block is issued. As a consequence, when the ancient block of a window gets empty it is no longer guaranteed that its "new" ancient block will be already at RAM, and as such it might be necessary to bring data from disk. Additionally, if the buffer pool is full, it will be also necessary to send a block to disk in order to open room for the upcoming block. The expiration process under the *SSM* scheme is described in detail in Procedure 3 (note that the procedure has a parameter to indicate which window the request comes from).

6.4.2 Discussion: I/O Load and Eviction Policy

The major advantage of *SSM* lies in a better use of memory space by avoiding that tuples are stored multiple times in the several window operators. This guarantees that no matter how many windows are defined over a given stream, the space cost will never exceed the size of the largest window in the set. As a consequence, *SSM* can handle a much greater number of queries than an unshared approach with the same amount of available memory before having to resort to secondary storage.

Procedure 3 expireOldestShared(*window_rank*)

let *ancient_block*: block at RAM holding soon-to-expire tuples of the window passed as argument

let *valid_index*: index of the oldest, non-expired tuple in the ancient block of the window passed as argument

```
 1:  ancient_block ← GETANCIENTBLOCK(window_rank);
 2:  valid_index ← GETVALIDINDEX(window_rank);
 3:  if ancient_block has only expired tuples then
 4:    if window_rank is the largest then
 5:      buffer_pool.REMOVE(ancient_block);
 6:    end if
 7:    new_AB ← GETNEXT(window_rank, ancient_block);
 8:    if new_AB is at buffer pool then
 9:      new_AB_Addr ← GETBPBLOCKADDRESS(new_AB);
10:      ancient_block ← buffer_pool.GET(new_AB_Addr);
11:    else
12:      new_AB_Addr ← GETDISKBLOCKADDRESS(new_AB);
13:      ancient_block ← READFROMDISK(new_AB_Addr);
14:      if buffer pool is full then
15:        victim_block ← GETVICTIMBLOCK();
16:        buffer_pool.SWAP(victim_block, ancient_block);
17:        WRITETODISK(new_AB_Addr, victim_block);
18:      else
19:        buffer_pool.ADD(ancient_block);
20:      end if
21:    end if
22:    SETANCIENTBLOCK(window_rank, ancient_block);
23:    valid_index ← 0;
24:  end if
25:  valid_index ← valid_index + 1;
26:  SETVALIDINDEX(window_rank, valid_index);
```

Now another important aspect is once the memory has been exhausted and access to disk is required, how much load *SSM* puts into the I/O subsystem. In order to determine that, consider that there are N windows of different sizes: $w_1 < w_2 < \ldots < w_N$. Let ρ be the likelihood of the next oldest block of a window being already at RAM after the current ancient block gets empty (see line 8 in Procedure 3)[15]. Assuming the buffer pool is full, the I/O pattern will be as follows: i) a write request will be issued every time a new recent block is created and ii) expiration of the ancient block of window w_i will incur, with likelihood (1-ρ): one read request, and, if $i < N$, one additional write request (for $i = N$, the ancient block is effectively removed from the buffer pool, and as such, there is no need to send data to disk to open room for the new ancient block). Algebraically:

$$\#IO = W + (1-\rho) \cdot [(N-1) \cdot (R+W) + R] \quad (6.5)$$

Note that in the limit, for ρ=0, the amount of I/O generated when processing the query set using *SSM* will be exactly the same as in *SlideM* (one pair of read and write request per window):

$$\#IO = W + (N-1) \cdot (R+W) + R = N \cdot (R+W)$$

This means that the shared execution mechanism will never perform more I/O than the unshared approach, and in the worst case the I/O pressure of the two schemes will be equivalent. For any ρ>0, *SSM* will reduce the amount of I/O, and the reduction will be as large as ρ. As discussed in previous section, the optimal eviction policy that maximizes ρ is the one that sends to disk the block that is not going to be needed for the longest time. For the single-window case, the choice is straightforward: the optimal victim block is always the most recently written block. This does not hold for multiple windows though, as the MRW block might be needed earlier by a small window than an intermediate block by larger windows. Instead, the optimal victim block in a multi-

[15] In fact, ρ represents the hit rate of the buffer pool.

window scenario can be determined by computing the distance – in number of blocks or time units – of the candidate blocks at the buffer pool to the ancient block of each window as follows: let d_{ki} be the distance of block k at BP to the ancient block of window w_i. For any block k, ref_k denotes the next time the block will be referenced by any window, and corresponds to the minimum value in the set of distances: $ref_k = \min\{d_{ki} \mid d_{ki} > 0\}$. The victim block v is the one with the maximum value for ref_k among the B candidates at buffer pool: $v = (k \mid ref_k = \max\{ref_1, ..., ref_B\})$.

This *distance-based* replacement policy creates clusters of blocks in the BP, immediately after the ancient block of each window as illustrated in Figure 6.9 below:

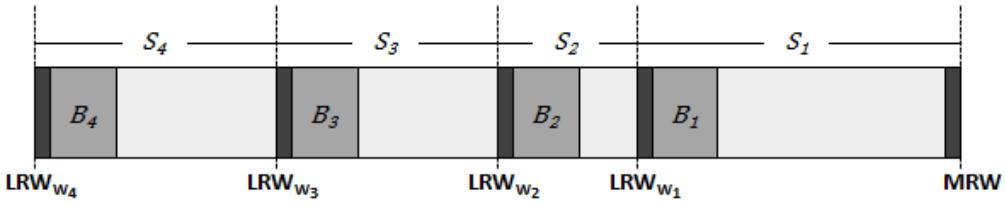


Figure 6.9: Arrangement of blocks at the buffer pool with SSM replacement policy.

Since each block is referenced only once by each window, the buffer pool hit rate of the scheme is given by the average percentage of blocks residing at memory of each segment:

$$\rho_{SSM} = \frac{1}{N}\sum_{i=1}^{N}(B_i/S_i) \tag{6.6}$$

where B_i is the number of memory-resident blocks of each segment and S_i is the corresponding total number of blocks.

Clearly, the more memory is available (larger B_i) and the more overlapping the windows are (smaller S_i), the higher the buffer pool hit rate ρ_{SSM} will be.

6.5 Experimental Evaluation

In this section we present an extensive experimental evaluation of the *SlideM* algorithm and the sharing scheme *SSM*. We have performed a wide variety of experiments with the objective of assessing:

1. *Effectiveness of SlideM in a real-world use-case*: we demonstrate the ability of the proposed algorithm in addressing memory shortages in a real scenario and compare its performance against the conventional memory-only implementations discussed earlier in this chapter (Section 6.5.2).

2. *High-performance nature of SlideM*: we examine *SlideM* performance under heavy load conditions. Results reveal that the algorithm was capable of handling very high input rates for multi-gigabyte windows while keeping latency under desirable levels (Section 6.5.3).

3. *Performance and scalability of SSM*: we show that the sharing mechanism *SSM* scales significantly better than an unshared approach (Section 6.5.4).

For the first set of experiments, we used queries and stream definitions taken from a real use-case. For the other two sets, we used synthetic queries and datasets. Tests setup and methodology are described next.

6.5.1 Setup and Methodology

We implemented our proposed techniques in Pulse [75], a Java-based stream processing engine from our industrial partner. Experiments were conducted on a server with two Intel Xeon E5420 2.50 GHz Quad-Core processors, 4 GB of RAM, and 4 SATA-300 disks distributed in two RAID-0 arrays, running Windows Server 2008 x64 and Hotspot x64 JVM (*configured with a 1 GB heap size*). One RAID array was used to host the OS while the other was used to hold window data during tests.

Measurements were taken as follows:

- A single Java application was responsible for generating, submitting and consuming tuples during the performance runs. Input data was submitted to the EP engine through local method calls using its API.

- Tests consisted in a *warm-up phase*, during which the EP system was brought to a steady state, and a subsequent *measurement interval* (MI), when the performance of the system was measured. The duration of both warm-up and MI was set to the time necessary for traversing 1.5 times the window – *e.g., an experiment with a 6-hour window ran for 18 hours (9h of warm-up plus 9h for MI).*

- We collected both application-level and system-level metrics. Average throughput was computed as the ratio between processed tuple count and elapsed time. Latency was computed through the `nanoTime()` method of the Java runtime, called immediately before and after sending tuples to the EP engine. Memory consumption was computed by the end of tests using standard Java SDK methods. CPU, disk, and process metrics were collected using the *System Monitor* tool of MS-Windows.

All experiments with *SlideM* and *SSM* used a fixed block size (64 kilobytes).

6.5.2 Call-Center Use-Case Results

Our first set of experiments mimics the workload conditions of a real event processing application, the call-center monitoring use-case we referred throughout this chapter. As mentioned earlier, the purpose of the application is to provide a real-time view of the operation of a large call-center chain. The company is spread over 20 geographical sites and has around 12,000 agents serving more than 3 million customer requests per day. A statistical module collects information about the calls and produces a stream of data items describing each step of the interactions between the call center and its customers. This data stream, whose schema is shown Figure 6.10, is then fed into the EP engine, where several KPIs are continuously computed.

```
AgentInteractions (
  timestamp                long,
  instance                 int,
  start                    long,
  sessionId                int,
  serviceId                int,
  agentId                  int,
  interactionLegId         int,
  alertingTime             int,
  busyTime                 int,
  wrapUpTime               int,
  waitTime                 int,
  direction                int,
  mediaId                  int,
  helpTime                 int,
  agentReleased            bool,
  mediaOutcome             int,
  finalSegment             bool,
  agentSite                int,
  callSite                 int,
  availableTime            int,
  availableTimeByService   int,
  held                     int,
  help                     int
)
```

Figure 6.10: Schema of the input stream in the call-center monitoring application

Overall, the application workload consists in processing a number of aggregation queries like Query 2 below:

Query 2: *Compute call-center statistics for the last 24 hours*

```
SELECT    COUNT(*),
          SUM(busyTime),
          AVG(busyTime)
FROM      calls [RANGE 24 HOURS SLIDE 10 SECONDS]
GROUP BY serviceId
```

On total, the application computes 144 aggregates, as the query above is applied to 6 different fields (*alertingTime*, *busyTime*, *wrapUpTime*, *waitTime*, *helpTime*, and *availableTime*), using 8 distinct grouping criteria (*instance*, *serviceId*, *agentId*, *mediaId*, *interactionLegId*, *agentSite*, *callSite*, and *direction*). Overall, since the COUNT aggregate can be shared by the queries with different fields, the engine is able to process those 144 aggregates using only 104 distinct SWA operators.

In our experimental evaluation we filled the tuples of the stream *AgentInteractions* with synthetic data since real datasets were not available due to confidentiality issues. The generated data, however, respected the critical properties of the original input stream, such as the cardinality of the attributes used as grouping key in the queries and the distribution of these groups over time. We did not replicate eventual oscillations on tuple arrival rate though, keeping the injection rate fixed in 1,000 tuples per second. All the tests were performed in a virtual machine with 8 cores, 2GB of RAM, and running Window Server 2008, as found in the production environment.

Results

We then compare the performance of memory-only SWA implementations against application performance when paging sliding window content to disk through the *SlideM* algorithm. Both the 1W and the 2LA SWA approaches were tested in each case. Results are presented in Figure 6.11.

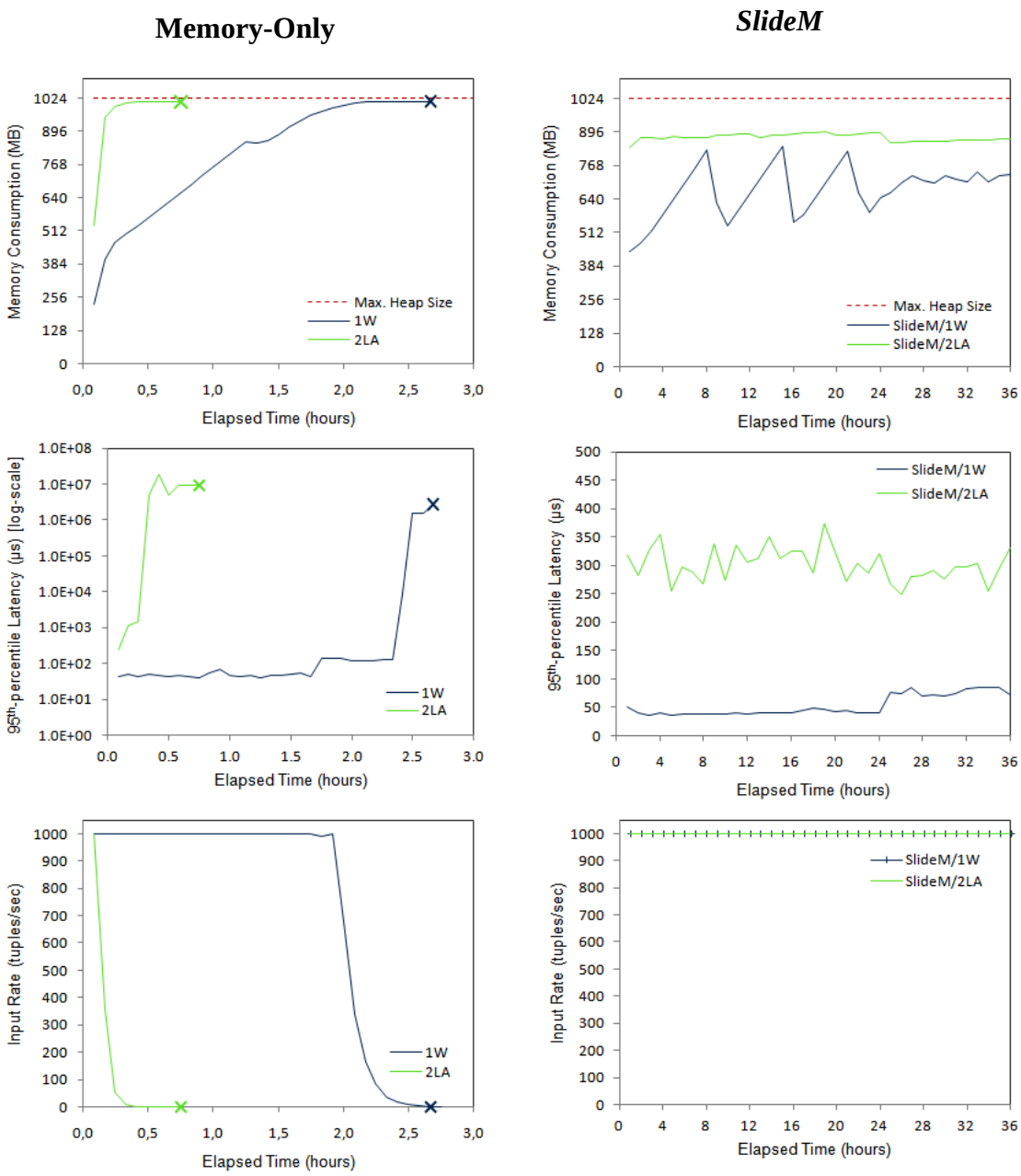


Figure 6.11: Performance of *SlideM* vs. memory-only implementations in real-world-based workload conditions (call-center use-case).

The three leftmost graphs show the performance of the two non-managed implementations, and illustrate what typically occurs with memory-constrained event processing applications in most Java-based EP engines: as the application working set approaches the available memory threshold, the system spends progressively more time with garbage collection, increasing the tuple processing latency and preventing the EP engine to cope with the data input rate. Since there is no data to be purged until the sliding window closes, the system eventually crashes with an out-of-memory error. In our tests this happened before 3 hours for the 1W implementation and before 1 hour when using the 2LA scheme, as signalized in Figure 6.11.

The results above contrast with application behavior when the *SlideM* algorithm is employed to manage the state of the sliding windows, as illustrated in the right part of Figure 6.11. Using our proposed algorithm allowed the experiments to complete without errors, while keeping performance metrics in desirable levels. As expected, memory consumption and tuple processing latency was larger when using the 2LA scheme than with the 1W approach in this use-case.

6.5.3 Performance of SlideM

As discussed before, many event processing applications, like those found in the financial trading environment, require that EP systems be able to process a considerable volume of data within very short periods of time. Using disks in these cases might be inadequate if they are not able to cope with the very high arrival rates and stringent latency requirements. In this section we examine how *SlideM* performs in such critical scenarios. For that, we employ a simple microbenchmark, which consists in computing one or more aggregations over a stock market data stream. Each input tuple has 4 attributes: *Timestamp*, *Symbol*, *Price* and *Volume* (about 28 bytes). We fill tuples by repeatedly cycling through a list of 100 stock symbols and assigning the tuple creation time to the timestamp field and random values to the other two. The workload consists in computing the volume-weighted average price (VWAP) of each stock over the last

hour, as shown in Query 3. Six runs of this experiment are performed, progressively scaling the injection rate from 50,000 up to 300,000 tuples per second.

Query 3: *Compute the VWAP of each stock over the last hour*

```
SELECT   Symbol, SUM(Volume*Price)/SUM(Volume)
FROM     Stock [RANGE 1 HOUR]
GROUP BY Symbol
```

Note that the window definition in the query above does not include a SLIDE, which means that the result must be updated whenever a new tuple arrives at the *Stock* stream[16]. In all experiments, the buffer size was set to the minimum, 2 blocks (128KB), so that system performance is always measured under maximum I/O pressure. Results are shown in Table 6.1

As we can see, the system was able to handle up to 300,000 tuples per second, with the CPU being the limiting factor at that point. Average processing latency was fairly low in all experiments (a few microseconds), and even the absolute maximum latency remained under acceptable levels as the load was increased. Disk utilization was also quite low in all runs, as it can be seen from the average disk queue length (ADQL) metric in Table 6.1. The reason is that the disk bandwidth required by *SlideM* at the maximum load of 300,000 tuples per second in this benchmark is only 16 MB/sec, which is still far from the maximum measured disk transfer rate, as discussed in section 6.3.1. This moderate load posed by *SlideM* into the I/O subsystem was crucial to remove a bottleneck (memory) without creating a new one, thus allowing the EP engine to fully exploit the available CPU power.

[16] An aggregation query without a SLIDE clause would probably make little sense if its result were to be output (i.e., used for monitoring purposes). In many cases, however, the result of an aggregation is used as input for further processing (e.g., pattern detection), and updated results must be produced as soon as new data is available.

Table 6.1: *SlideM* Performance, scaling injection rate

Injection Rate (tuples/sec)	Space Cost (GB)	Avg. Latency (ms)	Max. Latency (ms)	% CPU	ADQL
50,000	4.7	0.009	2.9	5%	0.016
100,000	9.4	0.012	3.6	13%	0.035
150,000	14.1	0.007	3.9	23%	0.056
200,000	18.7	0.012	16.7	37%	0.071
250,000	23.5	0.008	14.4	69%	0.103
300,000	28.2	0.008	18.9	100%	0.145

6.5.4 Performance of SSM

We now examine the performance of the shared execution scheme *SSM* and quantify to which extent it scales better than an unshared approach. Experiments here consist in processing N instances of Query 3, using either the original *SlideM* algorithm or its shared counterpart, *SSM*, under an input rate of 5,000 tuples per second. The number of queries in each experiment, N, took the following values: N = {2, 4, 8, 16, 32}. All N queries in the set have different window sizes, uniformly distributed in the interval [3600, 7200] seconds. Available memory was set to 512MB in all experiments (for the non-shared version, this amount was equally divided among the N buffer pools). We then measured for each algorithm the total space cost and the amount of pressure put onto the I/O subsystem. Results are depicted in Figure 6.12:

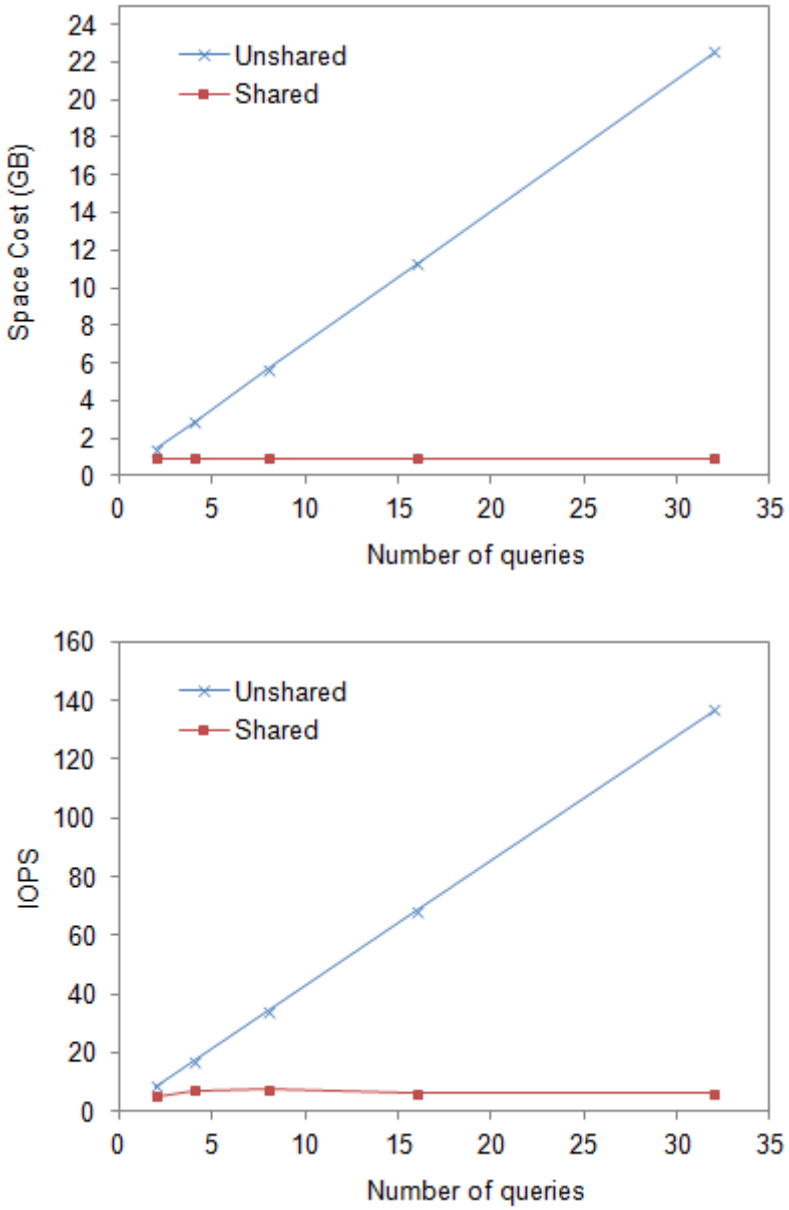


Figure 6.12: *SlideM* (unshared) vs. *SSM* (shared) in a multi-query scenario.

As expected, the total space cost (*memory and disk*) of the unshared approach grows linearly with the number of windows while with the shared strategy the space cost remains constant (*it is bounded to the size of the largest window*). *SSM* was also significantly more I/O-efficient, issuing up to 22 times less disk requests than the

unshared implementation. The explanation for this remarkable difference in the number of I/O requests between the two algorithms is that *SSM* benefits from the fact that adding more queries to the set reduces the distance between the overlapping windows, thus increasing the hit rate of the buffer pool as expressed in formula (6.6). This way, while the I/O pressure of the unshared implementation consistently grows as more windows are used, with *SSM* it tends to stabilize since the increase on buffer hit rate compensates for the increased number of simultaneous queries. The result is a much better scalability as we can see in Figure 6.12.

6.6 Related Work

There has been considerable work on resource management in stream processing systems [12] [40] [69]. For dealing with memory shortages, two approaches have been widely employed. The first consists in providing approximate answers by shedding load [28] [85] [98], with research on this area focusing essentially on minimizing error of approximations. However, many event processing applications rely on exact answers to perform complex data analysis and support real-time decision making. In these cases, techniques such as load shedding or approximation are not applicable. The alternative, then, is to use secondary storage as an extension of main memory. Indeed, such disk-based approach has been adopted in a number of proposals [21] [37] [61]. The focus of those works, though, is on processing of *join queries*. Liu et al [61] consider queries with multiple operators and propose strategies to choose which part of the operator states to spill during query execution in order to maximize the overall throughput. As discussed before, this is not an issue for sliding-window aggregates since the data access pattern can be accurately predicted. Farag and Hamad [37] propose a two-phase external-memory algorithm that joins the arriving tuples of one stream with the memory-resident data of the other streams, and postpones matching with the disk-resident portion until the stream runs out-of-space or arrival of new tuples stalls. Chakraborty and Singh [21] propose an *Exact Window Join* algorithm that deals with

memory shortages by deferring the load during high workload, and processing the deferred load during the period of low workload. This strategy, however, results in high delays (> 5 seconds) even for moderate data input rates (450 tuples per second). To the best of our knowledge, our work is the first to address the problem of exact answer computation of aggregations in memory-limited, high-throughput, environments.

Shared processing of sliding-window aggregates has been previously explored in a couple of proposals. Arasu and Widom [8] devise two algorithms for sharing execution of multiple sliding-window aggregates, where a common aggregation function is computed over different window sizes. These algorithms assume an *aperiodic* scenario, where results are produced on-demand (when user polls a query). Our proposed strategy, on the other hand, is for periodic aggregates and applies even when different aggregation functions are used. In [56] Krishnamurthy proposes a strategy for sharing the execution of multiple periodic sliding-window aggregates implemented under the *2LA* scheme. The strategy focuses on *computation sharing*, and consists in computing the partial aggregates with only one shared operator, rather than using one operator per query. It does not address, however, the *space sharing* problem introduced in this chapter, as the partial aggregates are still stored several times at the main window.

6.7 Summary

In this chapter we introduced techniques for overcoming the traditional memory limitations faced by event processing systems when processing aggregation queries over large sliding windows. We address the problem by proposing a novel buffer management algorithm, *SlideM*, which offloads sliding window state to disk during memory shortages. In order to further increase algorithm scalability, we also proposed *SSM*, a query sharing strategy that prevents explosion of space cost by storing the state of multiple overlapping sliding windows in a single, shared, repository. Experimental results demonstrated that the two techniques together provide significant performance

and scalability benefits. With *SlideM* the system was able to handle up to 300,000 events per second for multi-gigabyte windows while consuming only 128 kilobytes of main memory. In a scenario with multiple simultaneous queries, *SSM* reduced space cost by a factor of up to 24, issuing up to 22 less disk requests.

Chapter 7

Benchmarking EP Systems

The last three chapters have focused on performance analysis and optimization of event processing platforms. In this chapter, we move our attentions to *benchmarking*. We start by examining the unique challenges present in the development of an event processing benchmark and outlining possible approaches for addressing them. We then propose the *Pairs* benchmark for EP systems, and briefly review its tools. We finish the chapter by carrying out a comparative performance study of two event processing engines using *Pairs* as test case.

7.1 Introduction

Since the dawn of computing there has always been great interest in evaluating and comparing different systems with respect to their performance. Historically, these activities have been carried out with the help of *benchmarks*, synthetic programs that simulate the operations performed in a real environment, while collecting a series of metrics that characterize the performance of the target system. A well designed benchmark brings a number of benefits to its domain of application. First and foremost, it allows objective comparisons between existing systems, which usually serves as a stimulus for technology providers to improve their offerings. In addition, benchmarks are often used to assess the effectiveness of optimizations proposed in academia and

industry. Finally, they serve as reference for end-users to estimate the expected performance of candidate platforms on their production environments, assisting them in tasks like system sizing and capacity planning.

However, developing benchmarks for the event processing area is not a trivial task. In addition to the traditional challenges faced in any benchmarking initiative – e.g., need for representativeness, repeatability, resistance to benchmark specials, etc. – , the evaluation of EP systems imposes a number of other difficulties, such as the lack of standards and a very diversified application domain. In the remainder of this chapter we analyze in more details these challenges and discuss how we addressed them when developing the *Pairs* benchmark for EP systems. We introduce the benchmark dataset, workload, metrics, and execution rules, and then conduct a set of experiments involving two implementations of *Pairs* on real event processing engines.

7.1.1 Summary of Contributions

Overall, the main contributions of this chapter are:

- We review the key goals and major challenges for the development of an event processing benchmark and indicate approaches to tackle them (section 7.2).
- We propose the *Pairs* benchmark for EP systems (section 7.3) and introduce its toolkit (section 7.4).
- We present the results of an experimental evaluation using implementations of the *Pairs* benchmark on two popular event processing platforms (section 7.5).

7.2 Design Principles

In order to be useful, a benchmark should meet a number of quality requirements [43]: first, it should be *relevant*, that is, there must be a target audience interested in the information provided by the benchmark and confident that it is representative of its

application domain. Second, it should be *simple*, i.e., its specification should be easily understood by the general audience. It should also be *portable*, that is, it must not be bound to any specific system or architecture. Finally, it should be *scalable*, i.e., it must not pose any limitation for testing larger systems or loads. As we are going to see next, among those four quality attributes, relevance and portability are particularly challenging when dealing with event processing platforms.

7.2.1 Relevance

In order to be relevant, a benchmark must be **representative**, that is, it must realistically simulate how the target systems are used in their application domain. Thus, the design of a representative benchmark generally involves identifying a core set of operations frequently performed by a number of real applications, deriving their respective proportions, and reproducing them on the benchmark workload. Representativeness is arguably the most important attribute of *industry benchmarks* – those produced by standardization bodies like SPEC [88] and TPC [100]. Very often the performance information provided by those benchmarks is a compelling factor in customers purchasing decision, and consequently vendors tend to invest considerable resources in optimizing their solutions to achieve better results. It is therefore essential that a benchmark exercises the right operations, so that the invested resources actually improve the users experience. However, designing a general-purpose representative event processing benchmark is particularly challenging because there has generally been little information about how EP systems have been used in the real world. In spite of the several successful projects in the most diverse application domains, and the recent efforts in documenting use-cases by the Event Processing Technical Society (EPTS) [35], detailed and more concrete characterizations are still rare.

Another aspect of relevance is the **applicability** of a benchmark. Ideally, a benchmark should provide useful information for *all* users of its target technology. In the event processing context, the major obstacle for achieving this goal lies in the vast range of

domains where the technology has been employed, each with their own functional and performance requirements. For example, users in the capital markets are generally very concerned about processing latency, as short response times represent competitive advantage. Thus, sub-millisecond latencies are typically expected in the algorithmic trading domain. However, other applications, are generally not so latency-sensitive, and response times in the order of few milliseconds (e.g., fraud detection), seconds (e.g., traffic monitoring), or even minutes (e.g., supply-chain management) are acceptable. The several domains also have very different requirements in terms of volume of data, number of concurrent queries or query state size. These significant divergences makes virtually impossible for a single benchmark, with a single metric, to be representative of the entire spectrum of applications and provide all the information required by its heterogeneous target audience. The solution in this case might involve devising a set of smaller, domain-specific benchmarks, each with its own workload, dataset and metrics, or having a fully-customizable benchmark, like the SPECjms2007 [80], where users are able to configure and customize the workload accordingly to their requirements[17].

A final aspect of relevance is how **challenging** the workload of the benchmark is. Most EP systems are able to process very high volumes of data under certain workload conditions, as demonstrated in [86] and [103]. However, the test scenarios used in the studies so far are too simplistic, involving either a quite limited number of concurrent queries or very small queries states. Instead, a good benchmark should instigate vendors to implement state-of-the-art techniques that allow overcoming performance and scalability difficulties found by real users when implementing their applications.

[17] Note that even for customizable benchmarks a canonical setup still needs to be defined, so that comparable results can be produced and made generally available.

7.2.2 Portability

Portability has been a less concerning issue on many domains of benchmarking (e.g. databases, web servers), as industry standards like SQL and Java today facilitate the implementation of benchmarks on the most diverse platforms. This does not hold, however, in the event processing context, where the **lack of standard approaches and query languages** constitutes an important benchmark design challenge. As discussed in section 2.4, current EP engines have very different approaches for expressing event processing logic. Even when the systems follow the same design style, as it is the case with products like Esper, Streambase and Oracle, all using SQL-like query languages, often the syntax of one system is very different than that of another, and their features and capabilities also differ considerably. This not only makes hard to specify the benchmark in a precise and unambiguous way, but also complicates the implementation of the benchmark in different platforms.

In the next section we discuss how we addressed those issues in the context of the *Pairs* benchmark.

7.3 The *Pairs* Benchmark

The goal of *Pairs* is to assess the ability of EP platforms in processing increasingly larger number of continuous queries and event arrival rates while providing quick answers – three quality attributes equally important for an event processing engine. For that, the benchmark was specifically designed to meet a number of important requirements, while addressing some challenges, as discussed in previous section:

- *Relevance*: *Pairs* focus on an application domain where EP systems have been increasingly prevalent and for which performance is widely regarded as critical.
- *Representativeness/Comprehensiveness*: the workload scenario of *Pairs* is inspired on a real event processing use-case. In addition, the benchmark

workload exercises a core set of operations that appear repeatedly in most event processing applications and are offered in a way or another by all EP systems.

- *Challenging*: In order to excel on *Pairs*, EP systems will be required to have outstanding performance and scalability attributes, for instance by employing shared query processing techniques and gracefully adapting to changes.
- *Portability*: The workload of *Pairs* is specified in terms of high-level operations rather than a fixed set of queries to which EP systems must strictly adhere to. This allows the benchmark to be implemented on the most diverse platforms, in spite of all their functional and structural differences.
- *Configurability*: in order to minimize the effects of the vast range of EP application domains, the benchmark offers a great deal of customization, so that users can carry out experiments that resemble more closely their environments.

7.3.1 Scenario

The scenario for *Pairs* is an investment firm where a number of analysts interact with an enterprise trading system responsible for automating and optimizing the execution of orders in stock markets. Users of the system pose trading strategies which are continuously matched against live stock market data. The exercised trading strategies belong to a category broadly known in the financial domain as *statistical arbitrage* and consist in monitoring the prices of two historically correlated securities, looking for temporary digressions that indicate an opportunity to capitalize on market inefficiencies.

The general structure of the benchmark scenario, including the main entities and the corresponding cardinalities, is depicted in Figure 7.1.

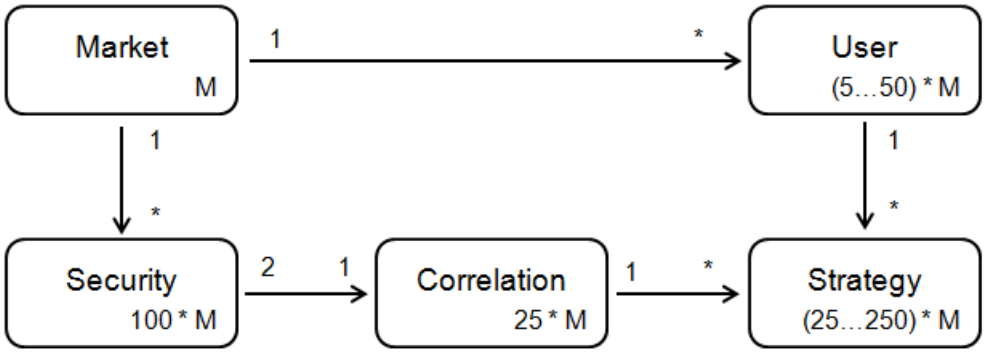


Figure 7.1: Overview of the *Pairs* benchmark scenario.

Per each one of the M stock markets, a number of securities (100) are monitored by the system, from which half are known to be mutually correlated (thus 25 correlations). Each of the users of the system manages exactly five strategies. The number of users per market ranges from five up to fifty, depending on the benchmark scale factor. In the basis case (5 users), there will be 25 strategies, each defined over a unique pair of correlated securities. On the limit, each pair of correlated securities on a given stock market is monitored by ten strategies of different users, each with its own parameters.

7.3.2 Input Data

Input data for the *Pairs* benchmark consists in a stream of simulated stock market data with the following schema:

StockTick (
  symbol : **string**,
  price : **int**,
  size : **int**,
  tickTS : **long**,
  TS : **long**
)

Figure 7.2: Input of the *Pairs* benchmark.

Each incoming tuple represents a trade operation executed in the stock market, such that *symbol* identifies the security being traded, *price* indicates the value, in cents, of the transaction, *size* represents the number of shares negotiated, *tickTS* is the time, in milliseconds, at which the trade has been executed (i.e., simulation clock time) and *TS* is the actual time the record was sent to the system under test (i.e., wall clock time)[18].

In the standard configuration, two hours of simulated market data is generated by a *data generator* application and submitted afterwards by a *driver* application to the system under test (SUT). For the sake of simplicity and understandability of results, all securities in the fictional market have the same update frequency, so the *symbol* attribute is filled by repeatedly cycling through a list of pre-generated Strings. The *price* in a tick is filled with data following a geometric brownian motion, a stochastic process widely used to model stock price behavior [6][103]. The *size* attribute is filled with random numbers, multiples of 10, uniformly distributed in the interval [100, 1000]. The *timestamp* is filled with the time the tick was generated, accordingly to the arrival pattern described next. The raw size of each tick tuple is 48 bytes.

Tick arrivals follow a Poisson process [6], with its λ parameter – which represents the average arrival rate – varying over time, resulting in an arrival pattern similar to the one illustrated in Figure 7.3. The reason for having a varying input rate is to simulate more realistically what happens in most real event processing applications, where new data arrives at different rates depending on the period of the day. Moreover, a varying input rate allows evaluating, with a single run, how the system responds to progressively larger loads.

[18] The *TS* field is used for computing response time and must be added to the records by the benchmark *test harness* during the performance runs.

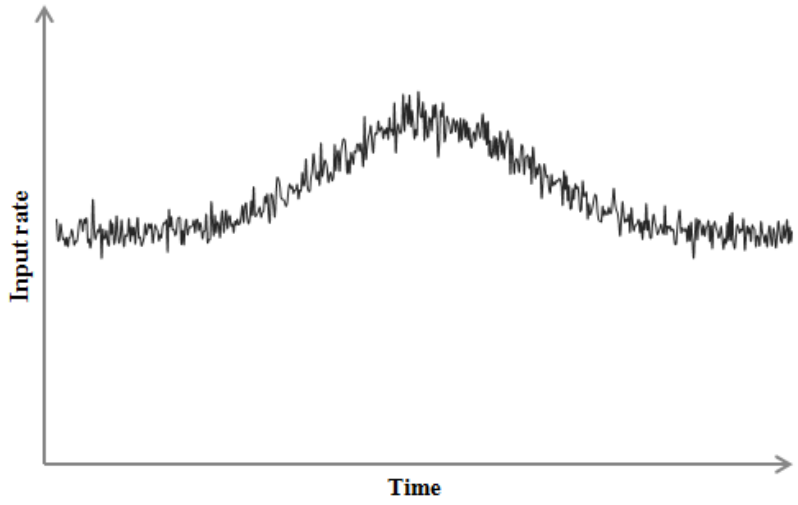


Figure 7.3: Input rate over time.

7.3.3 Workload

As mentioned earlier in this section, the benchmark workload consists in processing simultaneously a number of *Pairs* strategies. A *Pairs* strategy operates on the assumption that two securities are correlated and for this reason their prices tend to move together, as illustrated in Figure 7.4 (the chart shows the prices of two securities of a real stock exchange [19]). Eventually, though, oscillations in the market might make the prices to temporarily diverge. A *Pairs* strategy tries to identify these situations and react appropriately – for instance, by buying stocks from one security whose price remained stable when the price of the other security has risen – hoping that the prices will converge again soon. For that, the strategy makes use of a popular technical analysis tool called *Bollinger Bands*, computed over the ratio between the prices of the two securities (see Figure 7.5). By definition, the value of the ratio is *high* when it is above the upper band, and is *low* when it is below the lower band. A high ratio means that the first security is likely overvalued and/or the second security is probably undervalued. A low ratio means the opposite of that.

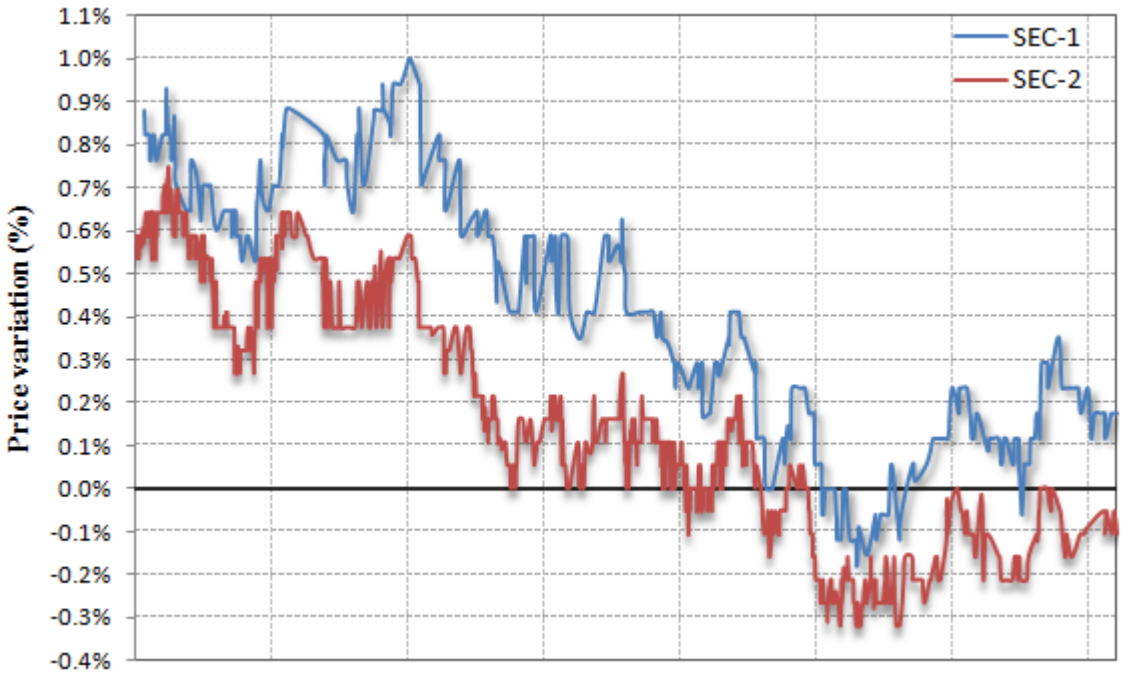


Figure 7.4: Price movement of two correlated securities.
(source: BM&F Bovespa [19])

In general, all strategies in the benchmark perform the same set of operations, described below, but each with different parameters:

1. *Compute indicators*: calculates the ratio and bands values that indicate the current state of correlation between the prices of the two monitored securities.

2. *Signal opportunities*: detects when the ratio crosses one of the bands.

3. *Position*: once a possible opportunity has been spotted, the system checks if it must change its current market position.

4. *Place orders*: if a change in market positioning is indeed required, the system must emit a pair of SELL and BUY orders. This step involves identifying the appropriate values for the parameters of each order (i.e., size and price).

5. *Manage risks*: once a market position has been assumed, it might be necessary to leave it sometime afterwards if the securities prices keep drifting apart, countering the expected reversal trend. The system must signal anytime price digression exceeds a given threshold and then react appropriately by emitting stop-loss orders.

Each of the steps above is discussed in further detail on next sections. Note that some of the operations will necessarily be replicated for each strategy running at the EP engine while for others sharing might be possible.

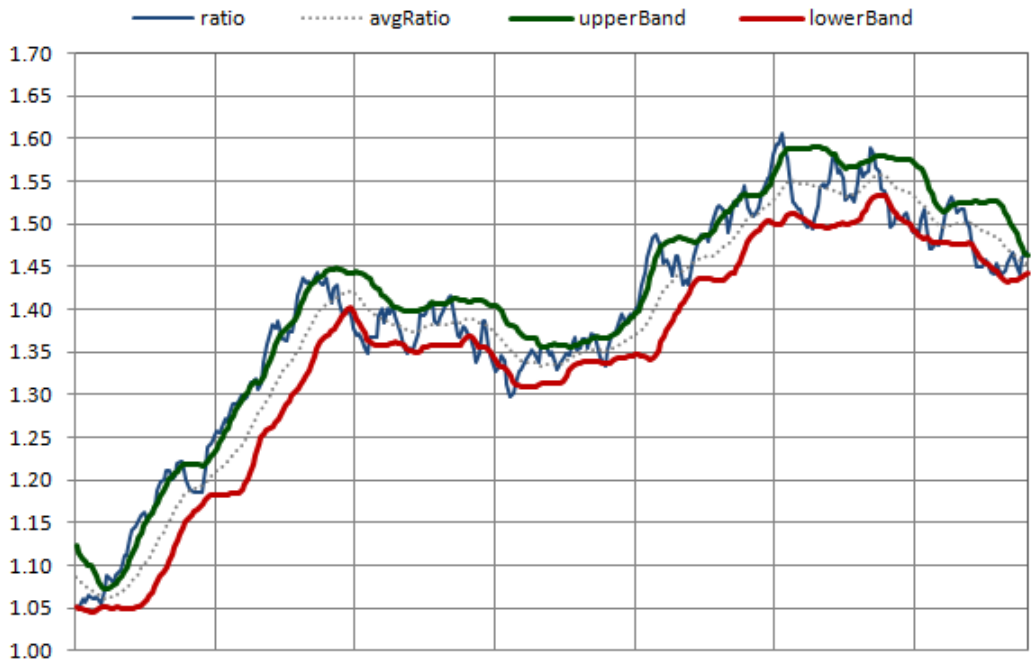


Figure 7.5: Indicators produced by a *Pairs* strategy.

Indicators Computation

The computation of indicators starts by filtering the incoming stock market data, letting pass only ticks from the two securities that are part of the strategy. Then, the prices of each symbol are aggregated over a given *time interval* (e.g., the average price during the last 10 seconds). These aggregates are then correlated to produce a *ratio*. Once more,

the last values of this ratio are aggregated over a *count-based window* (e.g. the last five tuples), finally producing the final metrics: the last value of the ratio, a moving average of the ratio and upper and lower bands (which correspond to the moving average plus the standard deviation multiplied by a positive and negative factor respectively). A schematic representation for the computation of indicators is illustrated in Figure 7.6.

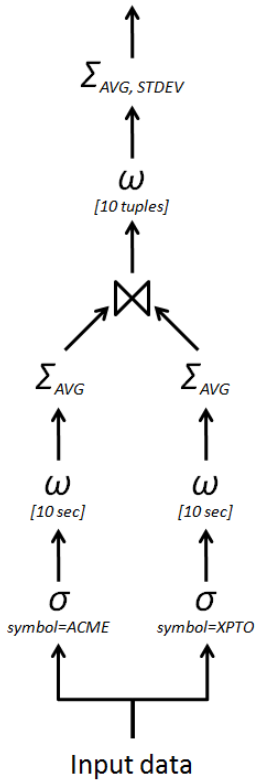


Figure 7.6: Indicators computation.

In order to ensure repeatability, the time window over which the two securities prices are initially aggregated is defined over the *tickTS* field of the input stream *StockTick*. In other words, the incoming ticks serve also as a clock in the benchmark simulation, so the concept of time passing is strictly associated with the arrival of new tuples. Note that by this definition, the aggregations must produce an updated result whenever the corresponding time window closes, which might happen even if the most recently arrived tick is not one of the two referenced in the strategy. Regarding the last sliding-

window aggregate operations (i.e., LAST, AVG, and STDEV over the ratio values), results must be emitted only after its count-based window is full (i.e., if the window size is 10, the aggregation must output a result if and only if the window contains 10 elements).

Opportunity Signaling

The values produced in the previous step are used to determine possible opportunities to capitalize on market inefficiencies. This happens when the line formed by the values of the ratio crosses either the lower or the upper band (see Figure 7.5), a condition expressed algebraically as:

$$(ratio(\tau_k) \geq Upper(\tau_k) \wedge ratio(\tau_{k-1}) < Upper(\tau_{k-1}))$$
$$\vee$$
$$(ratio(\tau_k) \leq Lower(\tau_k) \wedge ratio(\tau_{k-1}) > Lower(\tau_{k-1}))$$

where $ratio(\tau_i)$, $Upper(\tau_i)$, and $Lower(\tau_i)$ correspond respectively to the values of the ratio between the securities, the upper band and the lower band at the period τ_i.

Positioning

Whether the detection of a possible opportunity triggers the emission of orders or not depends on the current state of the strategy. More specifically, a strategy can be in three distinct states, namely: *flat*, *long-short*, *short-long*. In the *flat* state the strategy does not own any security. All the strategies start and finish the performance run at the *flat* state. In the other two states, the strategy holds a market position for one of the securities. For convention, *long-short* means that the strategy holds stocks from the first security and *short-long*, indicates that it owns shares from the second security. Figure 7.7 below illustrates the transitions between the states and their corresponding triggers.

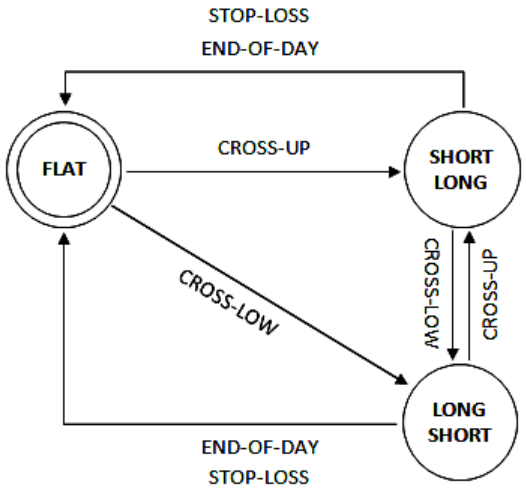


Figure 7.7: State machine of a *Pairs* strategy.

Order Placement

A transition from one state to another is completed only after the corresponding *BUY* and *SELL* orders have been emitted (if the strategy is currently on the FLAT state, only a *BUY* order for one of the securities is issued.). For that, the system must first determine the size and the price of each order. The price of both *SELL* and *BUY* corresponds to the price of the last trade executed in the market, for the securities in question. In practice, this means that the system must keep track of the last tick received for every security being monitored.

The size of the orders is determined by the amount of funds available for the strategy in question (in case of a *BUY* order) and the number of stocks currently owned (in case of a *SELL* order), which in turn must be maintained and constantly updated by the system as new orders are issued. The entire process of determining the sizes and prices of the orders is described in Procedure 4. For simplicity, the orders are assumed to be always accepted by the market and executed immediately.

Procedure 4 placeOrder()

let to*Sell*: the security to be sold
let *toBuy*: the security to be bought
let *sellGain*: amount of funds earned with the sell order
let *available*: total funds available

1: *sellPrice* ← GETLASTPRICE(to*Sell*);
2: *sellSize* ← GETSHARES(to*Sell*);
3: *sellGain* ← *sellSize* × *sellPrice;*
4: available = CURRENTBALANCE() + *sellGain*;
5: *buyPrice* ← GETLASTPRICE(to*Buy*);
6: *buySize* ← $10 \times \lfloor available \,/\, (buyPrice \times 10) \rfloor$;
7: SELL(*toSell*, *sellPrice*, *sellSize*);
8: BUY(*toBuy*, *buyPrice*, *buySize*);
9: UPDATEBALANCE(*available – buySize*);

Risk Management

Another condition that might trigger a change in a strategy state is when the prices keep drifting apart, countering the expected trend of reversal. If a strategy is currently positioned (i.e., either in the *long-short* or *short-long* states), the system must signalize this increase on market anomaly to prevent further losses. Again, the condition is expressed in terms of the *ratio* indicator computed in the first step:

$$ratio(\tau_{now}) \geq (1 + perc.) \times ratio(\tau_{positioning}) \quad \text{, when in the } \textit{short-long} \text{ state}$$

OR

$$ratio(\tau_{now}) \leq (1 - perc.) \times ratio(\tau_{positioning}) \quad \text{, when in the } \textit{long-short} \text{ state}$$

Where *perc* represents a percentage threshold, $ratio(\tau_{now})$ represents the current value of the ratio metric, and $ratio(\tau_{positioning})$ corresponds to the value of the ratio metric at the moment when the strategy took its current position.

When this situation is detected, the system must return to the *flat* state by emitting a *SELL* order for the currently owned security.

7.3.4 Output

The output of the *Pairs* benchmark consists in two event streams: `Indicator` and `MarketOrder`, whose tuples have the following forms:

Indicator (		***MarketOrder*** (	
strategy	: **string**,	*strategy*	: **string**,
ratio	: **double**,	*type*	: **string**,
avgRatio	: **double**,	*symbol*	: **string**,
upperBand	: **double**,	*price*	: **int**,
lowerBand	: **double**,	*size*	: **int**,
inputTickTS	: **long**,	*inputTickTS*	: **long**,
inputTS	: **long**	*inputTS*	: **long**
)		)	

Figure 7.8: Output of the *Pairs* benchmark.

The first represents the output of the first step in the strategy execution process and is used in the benchmark scenario for visualization and auditing purposes (*the stream serves to produce a graph like Figure 7.5 that allows users to better understand the decisions taken by the strategies*). The second stream represents the orders that were issued as a result of the execution of each strategy.

Tuples of the *Indicator* stream consist in a field *strategy*, indicating which strategy generated the result, and the fields *ratio*, *avgRatio*, *upperBand* and *lowerBand*, containing the values of the indicators described earlier. The *MarketOrder* stream consists in the fields *strategy*, again identifying the strategy that triggered the output, *type*, identifying the order as '*BUY*' or '*SELL*', and the fields *symbol*, *price* and *size*, which have the same meaning as in the input stream *StockTick*, and are computed as specified in the previous section. Besides the payload, tuples from both streams include two timestamps: *inputTickTS* and *inputTS*. Both are derived from the input event that triggered the emission of the output tuple and represent respectively the tick occurrence time (simulation clock) and its arrival time (wall clock). The former is used for

checking the correctness of the results while the latter is used for response time computation purposes.

7.3.5 Scaling

The workload of the *Pairs* benchmark is scaled by increasing the number of simultaneous strategies and, in some cases, the input rate. More specifically, the benchmark scale factor (SF) affects the number of users, and consequently the number of strategies executed in parallel as follows:

- Number of users: $5 \times SF$
- Total number of strategies: $25 \times SF$

Additionally, per every increment of ten in the scale factor, the basis input rate is incremented by 5,000 and the number of symbols is increased by 100 (this is to avoid too many similar strategies over the same symbols and to allow to assess how the system scales with changes in input rate and cardinality). The effect is as if a whole new market were now being monitored by a new team of analysts.

EXAMPLES:

- For a scale factor of 8, there will be 40 users, each managing 5 strategies, on a total of 200 strategies running in parallel on the trading system.
- For a scale factor of 15, there will be 75 users, each managing 5 strategies, on a total of 375 strategies running in parallel on the trading system, from which 250 are over the first set of 100 symbols and 125 are over the second set of 100 symbols.

While unconventional, this two-dimensional scaling scheme reflects more accurately what happens in stock markets (where tick arrival rates are directly related to the number of securities). In addition, the possibility of scaling the workload only by increasing the number of strategies, while keeping input rate or cardinalities fixed, allows assessing directly aspects like query scalability and resource sharing.

7.3.6 Measures

The performance of an event processing engine after a run of the *Pairs* benchmark is summarized using the p_{score} metric, which is defined as follows:

$$p_{score} = \frac{load}{99^{th}\ latency}$$

In the formula above, the term *load* represents the amount of computational work per unit of time that the benchmark poses to the SUT, and is a function of both the input rate and the number of concurrent strategies running at the engine – *these two are ultimately determined by the scale factor*. The denominator of the metric is the measured 99th-percentile processing latency, in seconds, for the tuples of the output event stream *MarketOrder*[19].

The intent of the metric above is to facilitate comparison among the several systems and benchmark runs. When defining the metric, we tried to benefit systems that are able not only to process high volumes of events, but also react quickly and scale well with respect to the number of concurrent queries. Therefore, in order to excel in *Pairs*, an event processing system must be able to:

i. Provide quick answers, and do that consistently;

ii. Handle increasingly larger loads (be it due to the number of simultaneous queries, input rate, or both).

Thus, a system A that does not reply as quickly as another B might have a lower score even if it manages to process more load. Also, if it replies quickly on average, but occasionally takes a long time to reply, it will also be penalized. Similarly, if a system

[19] . A detailed explanation on how the term *load* is computed and the rationale behind the p_{score} metric is presented in Appendix A.

replies very quickly, but only manages to achieve low scales factors, its score will hardly be outstanding.

Note that summarizing different performance aspects into a single number is always controversial, since different users have different perceptions on the value of each dimension depending on their requirements (e.g., for some, the best system is simply the one that replies faster, while for others it is the one that handles more load). Therefore, besides indicating the main metric, a *Pairs* report should include a number of other measures and information (e.g., number of strategies, input rate, average and maximum latency, latency histogram, etc.) to help users better understand the performance of the system under test and judge whether it fits their needs or not.

7.3.7 Execution Rules

Each run of *Pairs* starts with a short *ramp-up* phase (1 minute), during which the input rate progressively increases from zero up to its *peak value*[20]. The ramp-up is then followed by a 30-minute period where the input rates decreases until its *basis value*. After this period, the *measurement interval* (MI) of the benchmark run starts. The MI has a total duration of 1 hour, during which the input rate again increases to its peak value and then returns to its basis value. A final 30-minute period follows the MI, now with an increasing input rate. The several phases of the benchmark run are illustrated in Figure 7.9.

As mentioned before, the intent of this variation on the input rate is to observe how the performance of the SUT evolves across different load levels. Event processing applications run continuously for hours or even days without interruption, and as such it

[20] The purpose of the ramp-up is to give some time to the SUT for initializing its components and performing any JIT optimization on its code before handling the high event volumes.

is very likely that the conditions change during their execution. Gracefully responding to these load variations is therefore a fundamental quality that EP engines should possess. Furthermore, the shape of the event rate curve aims at simulating what typically happens in capital markets, where higher volumes of transactions are observed at market open and close, with sporadic peaks during the day. In the standard configuration of *Pairs*, the amplitude of the load variation is 1.5 (i.e. during *peak*, the input rate is 50% larger than the *basis* input rate).

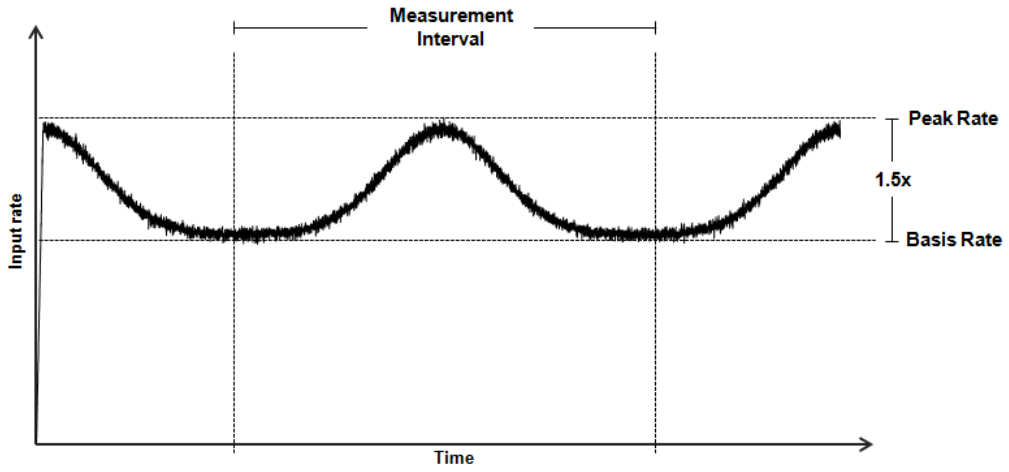


Figure 7.9: A *Pairs* benchmark run.

Note that for performance measuring purposes only the measurement interval is considered, but the SUT is required to produce correct answers for all the events received during the entire run.

7.3.8 Discussion: Is Pairs a good workload scenario?

There are a number of reasons why we believe the *Pairs* benchmark represents a good test case for EP platforms. First, the workload exercises several common features that appear repeatedly in most event processing applications, including:

1. *filtering* out ticks from securities which are not of interest;
2. *aggregating* events data over *temporal* and *count-based windows*;
3. *correlating* price data for interrelated securities;
4. *detecting patterns* from price movements;
5. *keeping track* and *updating* strategies' *state* upon the occurrence of certain events (position changes);
6. *performing lookups* to determine orders price and size;
7. *processing reactive rules* to determine which action must be taken when a opportunity or risk is spotted.

In addition, different from most benchmarks, which have a fixed set of queries (e.g.,[10]), the number of queries in *Pairs* increases with the system size. This is in conformance with what happens in many real event processing applications and allows evaluating important aspects like *query scalability* and *resource sharing*.

Other key benefits of *Pairs* are understandability and representativeness. The benchmark mimics a niche of application where event processing platforms have perhaps been most successful – capital markets. In fact, most products use simple financial use-cases to exemplify the usage of their features and languages in their documentation, so in principle it should be easy for anyone reasonably familiar with the area to understand *Pairs*. Moreover, *Pairs* is loosely based on a real use-case, and as such has a good chance to be representative of its domain of application.

It is also worth noticing that the benchmark offers a certain degree of freedom by not firmly specifying a set of queries to which EP platforms must strictly adhere to. The systems are free to make the best use of any of their individual features as long as they produce the correct answers. This is useful for addressing the portability issues discussed in section 7.2.

Finally, *Pairs* allows a great deal of customization. Users can control load intensity by setting high-level workload parameters like input rate and number of simultaneous strategies, or by altering scenario characteristics such as number of securities and configuration of the strategies. While the results obtained from these "customized" runs cannot be compared to standard runs, the ability to customize the workload enables users to exercise the systems in a manner closer to their own real environment.

7.4 Benchmark Implementation

The *Pairs* benchmark should be implemented and executed as illustrated in Figure 7.10 below:

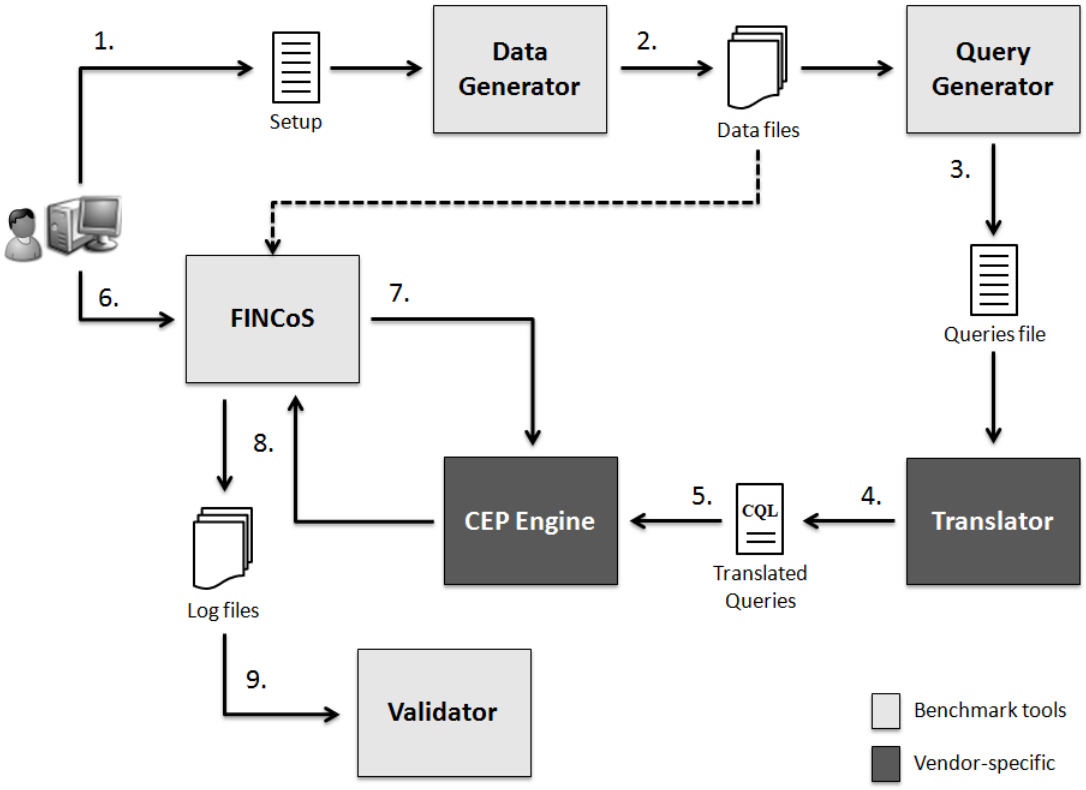


Figure 7.10: Benchmark execution flow.

1. The user specifies the workload parameters or, typically, uses the standard benchmark configuration;
2. A *data generator* application generates the benchmark input data files, containing the tuples of the *StockTick* stream, and an auxiliary file to be used by the query generator;
3. A *query generator* produces the benchmark workload and outputs it in a neutral representation (xml file);
4. A vendor-specific *translator* parses the file generated by the query generator and translates it into the query language used by the SUT;
5. The resulting queries/rules are loaded into the SUT;
6. The user starts the performance run;
7. A *test harness* (e.g., FINCoS framework) loads the generated data file(s) and submits the events on it to the SUT;
8. The SUT delivers results to the test harness;
9. A *validator* verifies the correctness of the answers produced by the SUT

Note that the benchmark infrastructure above enables users to run tests with real stock market data, thus allowing them to evaluate candidate platforms in a way that resembles more closely their production environments.

All the aforementioned tools are written in Java and require very little effort to be executed. The *Data Generator*, *Query Generator* and *Validator* applications are specific to the *Pairs* benchmark and are available for download at [16]. The FINCoS framework is benchmark-independent and can be downloaded from [38]. We describe each tool in further detail in Appendix B.

7.5 Experiments

In this section we present the results of a preliminary study where we implement the *Pairs* benchmark on two widely-used event processing platforms – one open-source and the other a developer version of a commercial product[21]. We acknowledge that the employed implementations may not be optimal, as they represent our own view on how the benchmark functional requirements could be met using the products available features. We recall, though, that the main goal of this section is to validate the *Pairs* benchmark, so we encourage researchers and vendors to create alternative implementations and disclose their numbers.

7.5.1 Setup and Methodolgy

All the tests were performed on a single 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.

Tests were conducted as indicated in section 7.4. Benchmark input data was submitted using the FINCoS framework. Likewise, the results produced by the target EP engines were received, processed and stored on disk by the framework. Latency measures were obtained by processing the *sink* output log file. Those observations were then saved into a database, from which we computed latency metrics (average, minimum, maximum, and 99th-percentile). System-level metrics like CPU utilization and memory consumption were collected using the *System Monitor* tool of MS-Windows

[21] The products are kept anonymous due to licensing restrictions. Throughout the rest of this section we refer to them as engines “X” and “Y”.

7.5.2 Results

Table 7.1 below shows the 99th-percentile latency and the corresponding p_{score} achieved by engines X and Y for scale factors ranging from 1 to 5.

Table 7.1: Results of the tests with *Pairs* on two event processing platforms.

Scale Factor	Input Rate† (evts/sec)	No. of Strategies	Metric/Engine			
			99th-percentile latency (ms)		p_{score}	
			X	Y	X	Y
1	**5,000**	**25**	63	22	47.62	136.36
2	**5,000**	**50**	175	24	34.29	250.00
3	**5,000**	**75**	203	-	44.33	-
4	**5,000**	**100**	351	-	34.19	-
5	**5,000**	**125**	91,862	-	0.16	-

† Basis input rate.

As it can be seen, engine Y performed considerably better than X, with its best result (250, for SF=2) outperforming the best result obtained by X (~48, for SF=1) by a factor of more than five. Interestingly, while engine Y had a better performance during tests, it was unable to execute the benchmark for scale factors above 2 – the system threw an exception during load phase (step 5 shown in Figure 7.10) indicating that the parsed application exceeded an limit of the Java environment (64KB method size). In contrast, engine X was able to run the benchmark without problems up to a scale factor of 4 – for SF=5 the system was most of the time overloaded, resulting in prohibitively high processing latencies, as it can be seen in Table 7.1.

7.5.3 Analysis

Besides allowing objective comparisons among different event processing platforms, another goal of *Pairs* is to serve as a relevant test case for analyzing the performance of the engines. In this section we carry out one such analysis taking engine X as example.

The results of our experimental evaluation revealed some interesting aspects about the performance of engine X. First, the CPU utilization across the tests with different scale factors presented an erratic behavior when compared to the expected load put into the system. As shown in Figure 7.11, the increase on CPU utilization was always very different from the linear growth expected for that particular range (i.e., from SF=1 to SF=5). The sub-linear increase from SF=1 up to SF=3 suggested at first that the engine were perhaps benefitting from the similarities among the running strategies via some computation sharing strategy. However, we could not confirm that hypothesis in the subsequent experiments, with scale factors of 4 and 5, since CPU utilization climbed at a much higher rate than expected at those points.

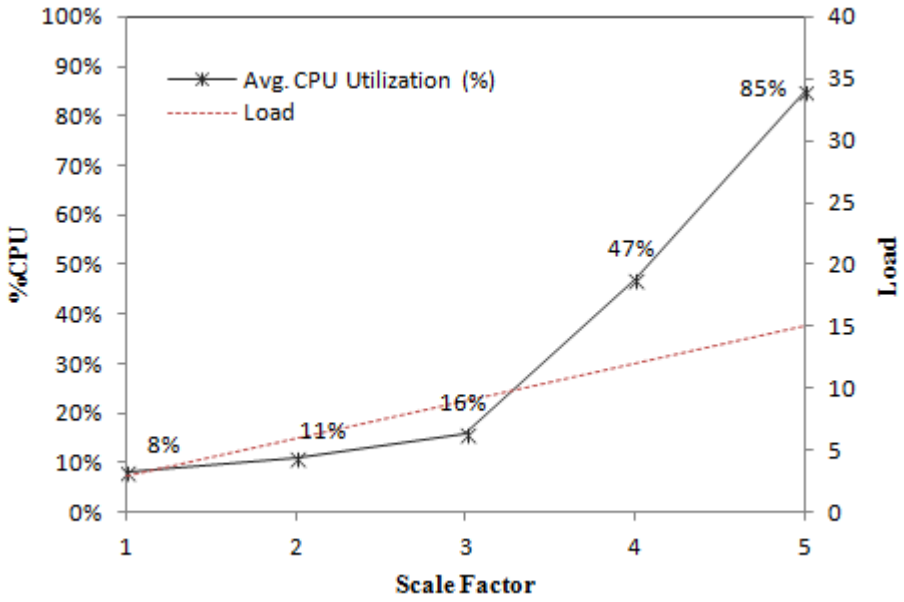


Figure 7.11: CPU utilization vs. benchmark load.

An odd CPU utilization pattern could also be observed in the course of a single benchmark run as illustrated in Figure 7.12. Not only was CPU utilization significantly higher in the second half of the experiment but it also more than doubled when injection rate was increased by a factor of only 1.5. The cause for the higher utilization at the

second half of the test seems to be on an increased garbage collection activity, probably due to larger query state sizes. It is not clear for us, however, why the CPU utilization increased at a higher proportion than the input rate – as a matter of fact, none of the two aforementioned patterns were observed in the tests with engine Y.

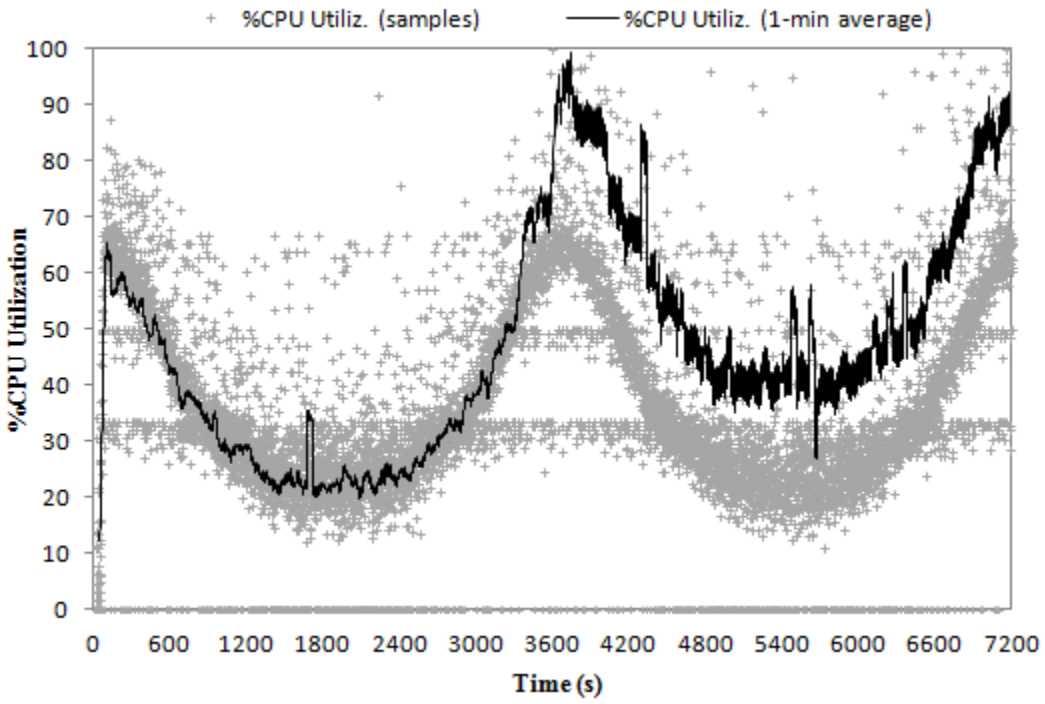


Figure 7.12: CPU utilization over time for engine X (SF=4).

Another interesting aspect to observe is how application performance, in particular, processing latency, is affected by the system state. Figure 7.13 shows the latency over time for the tests with scale factors 4 and 5. In the first graph, it is possible to clearly see some peaks in processing latency at the second half of the measurement interval, coinciding with the period of increased CPU utilization shown in Figure 7.12. The second graph illustrates the overload condition in which engine X executed during the experiments with a scale factor of 5, as mentioned in the previous section. As it can be seen, processing latency remained prohibitively high during almost half of the measurement interval, reaching a maximum of 92 seconds approximately 12 minutes

after the injection rate hit its peak. The system would then return to its normal latency levels only 10 minutes after that.

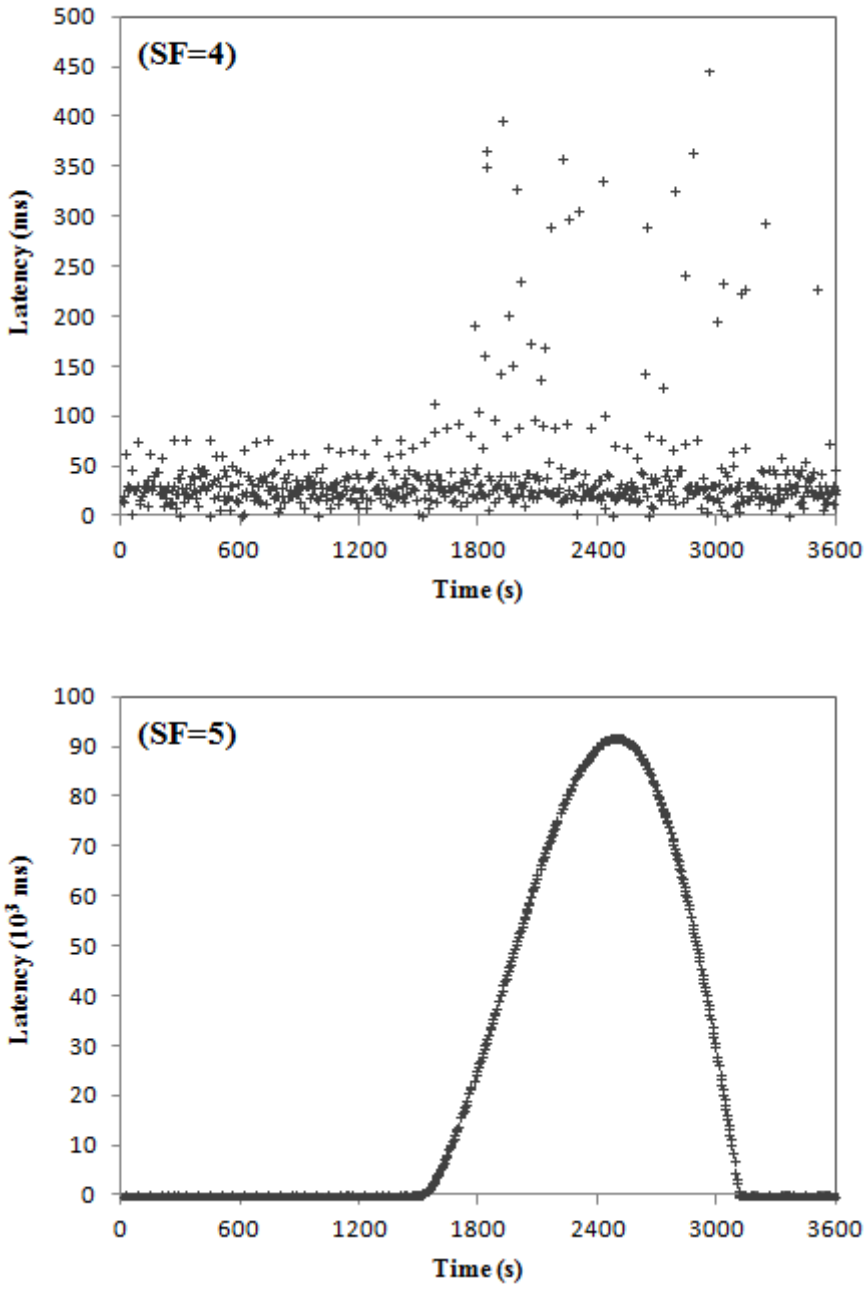


Figure 7.13: Processing latency over time, engine X.

7.6 Related Work

The performance of EP systems has been subject of much attention and debate. However, even though several numbers have been disclosed over the last years (e.g., [27], [33], [86], and [103]), there was, up to now, no obvious way to quickly and objectively compare the performance of the different offerings. To the best of our knowledge, the *Pairs* benchmark represents the first comprehensive work specifically targeted at addressing this gap. There have been, nonetheless, other benchmark proposals in related areas as well as ongoing initiatives in the event processing context itself.

Berndtsson et al [15] present the BEAST benchmark for active databases systems. The benchmark is designed to stress the performance-critical components of active database systems, including event detection, rule management, and rule execution. While BEAST was designed for active databases, it provides valuable insights on which aspects to focus on when assessing reactive behavior, and as such represents an important reference for evaluation of EP systems, particularly with respect to processing of event pattern rules.

In the context of data stream management systems, two research benchmarks have been proposed. The Linear Road benchmark [10] simulates a tolling system for a metropolitan area, where tolls for a given expressway are calculated based on congestion and accident proximity. Input data consists in a stream of position reports coming from simulated vehicles, over which a number of continuous and historical queries must be computed. The metric of the benchmark, L-Rating, represents the amount of load (measured in number of expressways) that a target system is able to handle while still meeting the established response time and correctness constraints.

Another benchmark for DSMS's, NEXMark [71], out of the Portland State University, simulates an online auction system where new items are continuously submitted for auction and new bids are continuously arriving. In spite of having been a work-in-

progress already for many years, NEXMark is an interesting complement to the more mature Linear Road benchmark, in that it provides more information about the systems under tests. Each of its eight queries was designed to test a specific operation provided by data stream engines (e.g. selection, aggregation, joins, windowing), which facilitates end-users to understand where exactly a given system is better than the other, and helps vendors to focus their optimization efforts.

The two aforementioned benchmarks are particularly relevant references as many event processing platforms of today have their roots on early academic projects in the data stream processing area. There are, however, a couple of issues that limit their adoption in the event processing context. First, being benchmarks designed for data stream management systems, they do not encompass the entire spectrum of functionality required by modern event processing applications – their workloads focus mainly on SQL-like operations, like selections, aggregations and joins, while touching very superficially (Linear Road) or not at all (NEXMark) features like event pattern detection and reactive behavior. In addition, the benchmarks do not measure important capabilities of EP engines, like the ability to adapt to changes in load conditions or share execution plans between similar queries.

Focusing on the communication side of event-driven applications, Sachs et al [80] introduce the SPECjms2007 benchmark for evaluating the performance and scalability of JMS-based messaging systems. SPECjms2007 allows measuring the performance of messaging systems in two distinct ways, using what has been called topologies. The horizontal topology evaluates the ability of messaging middlewares in handling increasing number of destinations (queues and topics), while keeping a fixed message traffic per location. The vertical topology, on the other hand, evaluates their ability in handling increasing message traffic while keeping fixed the number of destinations.

The *Securities Technology Analysis Center* (STAC) [82] is an industry consortium that focuses on the creation of standard methods for measuring the performance of trading systems. The benchmarks produced by the consortium cover a variety of technologies

used in capital markets, from messaging middlewares through tick databases. For this reason, STAC divides its benchmarks into three main categories: market data (*STAC-M* benchmarks), analytics (*STAC-A*), and execution (*STAC-E*). Each of these domains is further divided, based on the way financial firms buy products today. Of particular interest for the event processing community is the *STAC-A1* benchmark, which, according to STAC, has the purpose of *testing solutions that take inbound events from one or more sources, apply specific algorithms to those events, and generate outbound events* [87]. The benchmark, though, is not available to the general public and has been in development phase for several years, with no status updates since 2008.

7.7 Summary

In this chapter we introduced *Pairs*, a benchmark for evaluating and comparing the performance and scalability of event processing platforms. We started by reviewing the major challenges involved in the development of a benchmark for EP systems. We have seen that the diversity of products, lack of standards, and wide spectrum of application domains make it difficult to meet essential benchmark quality attributes, like relevance and portability. *Pairs* addresses the lack of common approaches and languages by not strictly specifying a set of queries, but rather the *operations* that must be performed – which in the end is what users are concerned with. This not only offers freedom for the products to implement the benchmark using their unique approaches, languages, and features, but also stimulates creative thinking on finding more efficient ways to solve the posed problems. The benchmark is also relevant, in that it exercises a set of operations that are present in almost all event processing applications. In addition, since it is based on a real use-case, chances are that its workload will mimic well how EP systems are used in the real-world. It is still hard to foresee, though, how general the benchmark is, and whether the information it provides will be valuable in other application domains. We believe, however, that the benchmark configurability properties might help to minimize this issue.

Having presented the benchmark specification, we then described how *Pairs* should be implemented on EP platforms and introduced its core tools. The benchmark toolkit is publicly available for download and includes a *data generator*, a *query generator*, and an *answer validation tool*. The benchmark also makes use of the FINCoS framework, as test harness, and a set of vendor-specific *translators*.

After implementing *Pairs* on two popular event processing platforms, we concluded this chapter by presenting the results of a comparative performance study. In our experiments, one of the engines managed to reach a maximum scale factor of 4, with processing latencies ranging from 1 up to 446 milliseconds (average: 49 ms; 99^{th}-perc.: 351 ms), obtaining a p_{score} of 34.19 – the best result though was obtained for a scale factor of 1 (p_{score} = 47.62), due to considerably lower latencies (99^{th}-perc.: 63 ms). The second engine achieved a significantly higher p_{score}: 250, with a 99^{th}-perc latency of 24 milliseconds, running at a scale factor of 2. However, it was unable to run the benchmark for scale factors above that. The tests also revealed some interesting facts about the performance of the first engine, like a regular increase of response time when faced with larger load levels and an odd variation on CPU utilization across the measurement interval

Chapter 8

Conclusions

Recent years have witnessed the consolidation of the event processing paradigm as an important research field and industrial trend. A number of projects were initiated at academic institutions, while tens of specialized startups appeared in industry, each offering their own event processing solution. EP systems then started to experiment increased adoption on the most diverse application domains, such as financial services, fraud detection, infrastructure management, business activity monitoring, and many others.

However, in spite of being often used in many mission-critical scenarios and having timeliness as one of their central compelling traits, until recently very little was known about the performance of event processing platforms. Only a few neutral scientific studies had been published, and there was a lack of common workloads and tools that allowed a unified approach for evaluating EP systems under comparable conditions.

This dissertation advanced the state-of-the-art by expanding the understanding on the performance of EP platforms via a series of experimental evaluations, and also by providing the instruments for others to conduct further studies and disclose more findings and results. Furthermore, a number of techniques aimed at improving the performance and scalability of event processing systems were proposed.

Specifically, in Chapter 3 we introduced FINCoS, a set of benchmarking tools for load generation and performance measuring of EP platforms. We then conducted an extensive performance study of three distinct systems in Chapter 4. Our experiments revealed some recurrent performance issues, which we addressed in the following chapters. In Chapter 5 we evaluated alternative data organization schemes (e.g., column-oriented) and proposed cache-aware algorithms to reduce memory consumption and improve the execution of continuous queries at the CPU. In Chapter 6 we addressed the problem of memory-constrained applications by introducing the *SlideM* buffer management algorithm and the *SSM* shared processing strategy. Finally, in Chapter 7 we described the *Pairs* benchmark for EP systems, and presented results for two engine implementations.

8.1 Contributions

In general, this dissertation provides the following practical contributions:

- *FINCoS*, a highly-configurable, scalable, and portable framework that the community can use to more rapidly evaluate the performance of event processing platforms, as well as to devise and experiment novel benchmarks.
- The *Pairs* benchmark, a comprehensive workload scenario and set of accompanying tools, which can be used to objectively assess and compare the performance of different EP systems.
- *SlideM*, an algorithm for managing the contents of very large sliding windows in memory-constrained scenarios, and its shared counterpart *SSM*, for processing multiple overlapping windows in a resource-efficient way.

In addition, this work offers a number of valuable insights regarding the performance of event processing platforms:

- In Chapter 4 we have verified that event processing platforms are indeed capable of handling very high input rates (*up to one-million events per second for simple filtering queries*), as emphasized in previous studies disclosed by vendors. However, we also noted that these figures vary dramatically depending on workload parameters, like window size and policy, tuple sizes, predicate selectivity, and cardinalities, and therefore much lower throughputs are typically achieved in practice. Results also exposed a poor utilization of memory resources by two of the three tested engines, which ended up causing trashing and out-of-memory failures in some tests. Finally, our study also revealed that the tested engines do not implement well-known optimization techniques such as *paning* for sliding-window aggregate operations or resource sharing when processing multiple similar queries.

- In Chapter 5, we have examined the impact of different data structures on the performance of two common event processing operations: moving aggregations over windowed event streams, and join of event streams with historic data. We have shown that a column-oriented organization scheme can outperform other widespread representations such as plain Java Objects, Object-arrays and key-value maps by considerable margins (*e.g., increases on query throughput by factors of up to 20, 34, and 272, respectively, in our aggregation tests*) and also reduce memory consumption considerably (*up to 67 times less space required*). We consider that these findings are valuable for both users implementing their event processing applications, which are now able to choose more wisely their event representation format, and vendors, which can enhance their products, for instance, by incorporating concepts from the column-oriented approach. In the same chapter, we have also identified a strong link between microarchitectural aspects and the performance degradation observed when query state grows. We have then proposed novel algorithms to minimize the losses caused by increased cache misses. The results were promising, with gains on query throughput ranging from 30% to 44% on our prototype.

- In Chapter 6 we have demonstrated that for some workloads it is possible to make use of disks (*thus saving memory resources*), even under very high event arrival rates (*hundreds of thousands of events per second*), without incurring in severe performance penalties (*e.g., processing latency was in the worst case below 20 milliseconds*).

8.2 Future Work

This dissertation covered several areas of the broad topic of event processing systems performance, including measurement tools, evaluation methodologies, experimental studies, and optimization techniques. Each of these areas constitutes a wide research space by itself and presents many interesting avenues for future work. Related to this particular work we can cite:

- *Measurement Tools*: the graphical nature of the FINCoS framework greatly facilitates the definition and monitoring of experiments, but might become troublesome for large experiment sets. Thus, we plan to extend the framework to support automated execution of performance runs.

- *Performance Analysis and Optimization*: we obtained promising results with the microarchitectural optimizations presented in Chapter 5, thus it would be interesting to delve more deeply into the topic and verify if EP systems in general can benefit from them in more diversified conditions. Likewise, the techniques proposed in Chapter 6 were designed to exploit the access pattern of sliding window operators during event arrivals/expirations, which allows excellent performance for aggregation queries. A natural direction for future work would then be to investigate state-spilling mechanisms that work well for a more diversified gamma of operations (e.g., joins or pattern matching). Also, the results presented in that chapter were obtained with conventional hard drives. It shall be interesting to observe the behavior of the proposed techniques in

conjunction with faster storage technologies such as solid-state disks (SSDs) and phase-change memories (PCMs).

- *Benchmarking*: another interesting avenue for future work is to conduct more studies using the *Pairs* benchmark on additional platforms and under more varied (non-standard) conditions. Also, given the wide range of application domains where EP systems have been employed and the strong tendency for specialization that has been observed recently, we believe that *Pairs* will unlikely solve completely the lack of benchmarks. We consider that this diversity of domains will eventually require the development of additional benchmarks and hope that our work serves as inspiration for future research in this area.

Bibliography

[1] Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S.B.: Aurora: a New Model and Architecture for Data Stream Management. *VLDB Journal*, Vol. 12, August 2003, 120-139.

[2] Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD* (Vancouver, Canada), 967-980.

[3] Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient Pattern Matching over Event Streams. In *Proceedings of the 2008 ACM SIGMOD* (Vancouver, Canada), 147-160.

[4] Ailamaki, A., DeWitt, D.J., Hill, M.D., Wood, D.A.: DBMSs On A Modern Processor: Where Does Time Go? In *Proceedings of the 25th VLDB Conference* (Edinburgh, Scotland), 1999, 266-277.

[5] Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving Relations for Cache Performance. In *Proceedings of the 27th VLDB Conference* (Rome, Italy, 2001) 169-180.

[6] Aldridge, I.: High-frequency trading: a practical guide to algorithmic strategies and trading. Wiley trading series. ISBN 978-0-470-56376-2.

[7] Aniello, L., Baldoni, R., Querzoni, L.: Adaptive Online Scheduling in Storm. In *Proceedings of DEBS 2013*, 207-218.

[8] Arasu, A., Widom, J.: Resource Sharing in Continuous Sliding-Window Aggregates. In *Proceedings of the 30th VLDB Conference* (Toronto, Canada, September 2004), 336-347.

[9] Arasu, A., Babu, S., and Widom, J.: The CQL continuous query language: semantic foundations and query execution. *VLDB Journal* Vol. 15, Issue 2, 2006, 121-142.

[10] Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear Road: A Stream Data Management Benchmark. In *Proceedings of the 30th VLDB Conference* (Toronto, Canada, 2004), 480-491.

[11] Arasu, A., et al.: STREAM: The Stanford Stream Data Manager. In Proceedings of SIGMOD 2003.

[12] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J.: Models and Issues in Data Stream Systems. In *Proceedings of PODS 2002*, 1-16.

[13] Babu, S., Bizarro, P.: Adaptive Query Processing in the Looking Glass. In *Proceedings of CIDR 2005*, 238-249.

[14] Belady, L.A.: A Study of Replacement Algorithms for Virtual-Storage Computer. In Proceedings of IBM Systems Journal. 1966, 78-101.

[15] Berndtsson M., Geppert A., Lieuwen D., Roncacio, C.: 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.

[16] BiCEP project web site: http://bicep.dei.uc.pt

[17] Bizarro, P., et al.: Event Processing Use Cases. Tutorial, DEBS 2009, Nashville USA.

[18] Bizarro, P.: BiCEP - Benchmarking Complex Event Processing Systems. In *Proceedings of the 1st Dagstuhl Seminar on Event Processing*, November 2007. Available at: http://drops.dagstuhl.de/portals/index.php?semnr=07191 Accessed: June 2013.

[19] BM&F Bovespa stock exchange web site, http://www.bmfbovespa.com.br

[20] Botan, I., Derakhshan, R., Dindar, N., Haas, L.M., Miller, R.J., Tatbul, N.: SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. In *Proceedings of PVLDB 2010*, 232-243.

[21] Chakraborty, A., and Singh, A.: Processing Exact Results for Sliding Window Joins over Time-Sequence, Streaming Data Using a Disk Archive. In

Proceedings of the 1st Asian Conference on Intelligent Information and Database Systems (Vietnam 2009), 196-201.

[22] Chakravarthy, S, Mishra, D.: Snoop: An Expressive Event Specification Language for Active Databases. Data Knowl. Eng. (DKE) 14(1):1-26 (1994).

[23] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah., M.A.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of CIDR 2003*.

[24] Chandy, M.K., Etzion, O., von Ammon, R.: The event processing manifesto. In *Proceedings of Second Dagstuhl Seminar on Event Processing*, March 2011.

[25] Cugola, G., Margara, A.: Processing Flows of Information: From Data Stream to Complex Event Processing. In *Proceedings of ACM Computing Surveys* (CSUR), Volume 44 Issue 3, Article No. 15. June 2012.

[26] Dayarathna, M., and Suzumura, T.: A Performance Analysis of System S, S4, and Esper via Two Level Benchmarking. In *Proceedings of the 10th International Conference on Quantitative Evaluation of Systems* (Buenos Aires, Argentina, 2013), 225-240.

[27] Dekkers, P.: Complex Event Processing. *Master Thesis* Computer Science, Radboud University Nijmegen, Thesis number 574, October 2007.

[28] Dobra, A., Garofalakis, M., Gehrke, J., Rastogi, R.: Processing Complex Aggregate Queries over Data Streams. In *Proceedings of the 2002 ACM SIGMOD*, Madison, Wisconsin, USA.

[29] Drools Fusion web site, http://www.jboss.org/drools/drools-fusion.html

[30] Eaton, C., Deroos, D., Deutsch, T., Lapis, G., Zikopoulos P.: Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data. McGraw-Hill, 2012.

[31] Eckert, M.: Complex Event Processing with XChange EQ: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events. *PhD thesis*, Institute for Informatics, University of Munich, 2008

[32] Eckert, M., Bry, F.: Complex event processing (CEP). In *Proceedings of Informatik Spektrum*, 2009, 163–167.

[33] Esper Performance. Available at http://docs.codehaus.org/display/ESPER/Esper+performance. Accessed on June 2013.

[34] Esper web site, http://esper.codehaus.org/

[35] Event Processing Technical Society (EPTS), http://www.ep-ts.com

[36] Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications (2010), 1-360.

[37] Farag, F., and Hamad M.A: Adaptive Execution of Stream Window Joins in a Limited Memory Environment. In *Proc. of the 11th International Database Engineering and Applications Symposium* (Banff, Canada, 2007), 12-20.

[38] FINCoS Framework web site: https://code.google.com/p/fincos/

[39] Fülöp, L.J., Toth, G., Rácz, R., Pánczél, J., Gergely, T., Beszédes, A., Farkas, L.: Survey on Complex Event Processing and Predictive Analytics. *Technical Report*, University of Szeged, Department of Software Engineering, July 2010.

[40] Golab, L., Özsu, M. T.: Issues in Data Stream Management. *SIGMOD Record*, Vol. 32, Issue 2, 5–14 (2003).

[41] Grabs, T., Lu, M.: Measuring Performance of Complex Event Processing Systems. In *Proceedings of TPCTC 2011*, 83-96.

[42] Gray, J, et al.: Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. In *Proceedings of Data Mining and Knowledge Discovery*, Vol. 1, Issue 1, 29-53 (1997).

[43] Gray, J. (editor): The Benchmark Handbook for Database and Transaction Processing Systems, 2nd Edition. Morgan Kaufmann, 1993.

[44] Gualtieri, M., Rymer, J.R.: The Forrester Wave™: Complex Event Processing (CEP) Platforms, Q3 2009.

[45] Guerra, D., Gawlick, U., Bizarro, P.: An Integrated Data Management Approach to Manage Health Care Data. In *Proceedings of DEBS 2009*.

[46] Gyllstrom, D., Wu, E., Chae, H., Diao, Y., Stahlberg, P., Anderson, G.: SASE: Complex Event Processing over Streams. CoRR (2006).

[47] Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufman Publishers, 4th edition, 2007.

[48] Hinze, A., Sachs, K., Buchmann, A.P.: Event-based applications and enabling technologies. *In Proceedings of DEBS 2009*, Art. 1, 15 pages.

[49] Hirzel M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A Catalog of Stream Processing Optimizations. Technical Report, IBM Research Division and New York University, September 2011.

[50] Huppler, K.: The Art of Building a Good Benchmark. In Proc. of TPCTC 2009.

[51] Intel VTune: http://software.intel.com/en-us/intel-vtune/

[52] Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., Venkatramani, C.: Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *SIGMOD Conference* (2006) 431-442.

[53] Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Çetintemel, U., Cherniack, M., Tibbetts, R., Zdonik, S.B.: Towards a streaming SQL standard. *In Proceedings of PVLDB 2008*, 1379-1390.

[54] Jain, R.: The Art of Computer Systems Performance Analysis. John Wiley and Sons, New York, 1991.

[55] Kitsuregawa, M., Tanaka, H., Moto-Oka, T.: Application of Hash to Data Base Machine and Its Architecture. New Generation Computing 1 (1): 63–74, March 1983.

[56] Krishnamurthy, S.: Shared Query Processing in Data Streaming Systems. *Ph.D. Thesis*, University of California, Berkeley, 2006.

[57] Kulkarni, D., Ravishankar, C.V., Cherniack, M.: Real-time, load-adaptive processing of continuous queries over data streams. In *Proceedings of DEBS 2008*, 277-288.

[58] Lakshmanan, G.T., Rabinovich, Y.G., Etzion, O.: A Stratified Approach for Supporting High Throughput Event Processing Applications. In *Proceedings of DEBS 2009*, Article No. 5.

[59] Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P.A.: No Pane, no Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. SIGMOD Record (2005), Vol. 34, Issue 1, 39-44.

[60] Liang, S., Fodor, P., Wan, H., Kifer, M.: OpenRuleBench: An Analysis of the Performance of Rule Engines. In *WWW (2009)*, 601-610.

[61] Liu, B., Zhu, Y., and Rundensteiner, E.A.: Run-Time Operator State Spilling for Memory Intensive Long-Running Queries. In *Proceedings of the 2006 ACM SIGMOD*, (Chicago, Illinois, USA), 347-358.

[62] Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, 2002. ISBN: 0-201-72789-7.

[63] Mendes, M.R.N., Bizarro, P., Marques, P.: A Framework for Performance Evaluation of Complex Event Processing Systems. In *Proceedings of DEBS 2008*, 313-316.

[64] Mendes, M.R.N., Bizarro, P., Marques, P.: A Performance Study of Event Processing Systems. In *Proceedings of TPCTC 2009*, 221-236.

[65] Mendes, M.R.N., Bizarro, P., Marques, P.: Assessing and Optimizing Microarchitectural Performance of Event Processing Systems. In *Proceedings of TPCTC 2010*, 216-231.

[66] Mendes, M.R.N., Bizarro, P., Marques, P.: Towards a Standard Event Processing Benchmark. *In Proceedings of ICPE 2013*, 307-310.

[67] Mendes, M.R.N., Bizarro, P., Marques, P.: Overcoming Memory Limitations in High-Throughput Event-Based Applications. In *Proceedings of ICPE 2013*, 399-410.

[68] Mendes, M.R.N., Bizarro, P., Marques, P.: FINCoS: benchmark tools for event processing systems. In *Proceedings of ICPE 2013*, 431-432.

[69] Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G.S., Olston, C., Rosenstein, J., Varma, R.: Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *Proceedings of CIDR 2003* (Asilomar, California, USA).

[70] Nambiar, R., Poess, M.: The Making of TPC-DS. In VLDB (2006) 1049-1058.

[71] NEXMark Benchmark, http://datalab.cs.pdx.edu/niagara/NEXMark

[72] Oracle OEP web site, http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html

[73] Owens, T. J.: Survey of event processing. *Technical report*, Air Force Research Laboratory, Information Directorate, December 2007.

[74] Progress Apama web site: http://www.progress.com/en/apama/

[75] Pulse: http://www.feedzai.com/products

[76] Rabinovich, E., Etzion, O. Gal, A.: Pattern Rewriting Framework for Event Processing Optimization. In *Proceedings of DEBS 2011*, 101-112.

[77] Ramamurthy R., and DeWitt, D.J.: Buffer-pool Aware Query Optimization. In *CIDR 2005*, 250-261.

[78] Roy, B.V.: A Short Proof of Optimality for the MIN Cache Replacement Algorithm. In Proceedings of Information Processing Letters, 2007, 72-73.

[79] RuleCore web site, http://www.rulecore.com/

[80] Sachs, K., Kounev, S., Bacon, J., Buchmann, A.P.: Workload Characterization of the SPECjms2007 Benchmark. In Proceedings of EPEW07, volume 4748, 228-244, Springer, 2007.

[81] Seagate Cheetah hard disk Data Sheet. Available at http://www.seagate.com/files/docs/pdf/datasheet/disc/cheetah-15k.7-ds1677.3-1007us.pdf. Accessed on July 2013.

[82] Securities Technology Analysis Center (STAC) Benchmark Council, http://stacresearch.com/council

[83] SPECjsm2007 benchmark: http://www.spec.org/jms2007/

[84] SPEC Research Group (RG) tools repository: http://research.spec.org/tools.html

[85] Srivastava, U., and Widom, J.: Memory-Limited Execution of Windowed Stream Joins. In *Proceedings of the 30th VLDB Conference* (Toronto, Canada, September 2004), 324-335.

[86] STAC Report: Aleri Order Book Consolidation on Intel Tigertown and Solaris 10. Available at: http://www.stacresearch.com/node/3844. Accessed on June 2013.

[87] STAC-A1 Benchmark, http://www.stacresearch.com/a1

[88] Standard Performance Evaluation Corporation (SPEC), http://www.spec.org

[89] Stanford RAPIDE Project web site, http://complexevents.com/stanford/rapide/

[90] Stonebraker, M., Abadi, D.J., et al.: C-Store: A Column-Oriented DBMS. In Proc. of the 31st VLDB Conference, Trondheim, Norway, 2005.

[91] Stonebraker, M., Çetintemel, U., Zdonik, S.B.: The 8 requirements of real-time stream processing. In Proceedings of SIGMOD Record. 2005, 42-47.

[92] Storm web site, http://storm-project.net/

[93] Stream Query Repository, http://infolab.stanford.edu/stream/sqr/

[94] Streambase web site, http://www.streambase.com/

[95] StreamCruncher web site, http://www.streamcruncher.com/

[96] Svensson, M.: Benchmarking the performance of a data stream management system. Master Thesis Computer Science, Uppsala Universitet, November 2007.

[97] Sybase Event Stream Processor web site http://www.sybase.com/products/financialservicessolutions/complex-event-processing

[98] Tatbul, N., Etintemel, U., Zdonik, S. B., Cherniack, M., and Stonebraker M.: Load shedding in a Data Stream Manager. In *Proceedings of the 29th VLDB Conference* (Berlin, Germany, September 2003), 309-320.

[99] TIBCO Business Events web site, http://www.tibco.com/products/event-processing/complex-event-processing/businessevents/default.jsp

[100] Transaction Processing Performance Council (TPC), http://www.tpc.org

[101] Tucker, P., Tufte, T., Papadimos, V., Maier, D.: NEXMark – A Benchmark for Queries over Data Streams. Technical Report, OGI School of Science & Engineering at OHSU, September 2008.

[102] Wahl, A., & Hollunder, B.: Performance Measurement for CEP Systems. In *Proc. of the 4th International Conferences on Advanced Service Computing* (Nice, France, July 2012), 116-121.

[103] White, S., Alves, A., Rorke, D.: WebLogic event server: a lightweight, modular application server for event processing. In *Proceedings of DEBS 2008*, 193-200.

[104] Wu, E., Diao, Y., Rizvi, S.: High Performance Complex Event Processing over Streams. In *Proceedings of SIGMOD 2006*.

[105] Yang, Q., Koutsopoulos, H.N.: A Microscopic Traffic Simulator for Evaluation of Dynamic Traffic Management Systems. Transportation Research C, 4(3):113–129, June 1996.

[106] Zeitler, E., Risch, T.: Massive Scale-out of Expensive Continuous Queries. PVLDB (2011) 1181-1188.

[107] Zukowski, M., Boncz, P.A., Nes, N., Heman, S.: MonetDB/X100 - A DBMS In The CPU Cache. IEEE Data Engineering Bulletin, Vol. 28, No. 2: 17-22 (June 2005).

[108] Zukowski, M., Heman, S., Nes, N., Boncz, P.A.: Super-Scalar RAM-CPU Cache Compression. In Proc. of the 22nd ICDE, Atlanta, USA, 2006.

Appendix A

The p_{score} Metric

In this section we discuss the rationale behind the metric of the *Pairs* benchmark, p_{score}, whose purpose is to facilitate comparison among different benchmark runs and systems. As mentioned earlier, the metric takes into consideration both the amount of load posed by the benchmark workload and the speed of the responses produced by the SUT. The metric has been chosen as to be fair. More specifically, it had to present two properties:

i. If two systems manage to handle the same load (i.e., same scale factor), the ratio between their scores must be exactly the ratio between their processing latency;

ii. If two systems present the same processing latency, the ratio between their scores must be exactly the ratio between the load they handled.

Note, however, that due to the way the benchmark scales, the load to which the SUT is submitted does not increase linearly with the scale factor. For instance, going from a scale factor of 9 to 10 has the only effect of adding 25 more strategies over the same set of symbols. On the other hand, going from a scale factor of 10 to 11 not only adds 25 more strategies, and over a whole new set of symbols, but also increases the benchmark basis input rate by 5,000 events per second. Table A.1 below summarizes the differences in the workload parameters for the aforementioned scale factors:

Table A.1: Workload parameters for different scale factors

	Scale Factor		
Parameters	**9**	**10**	**11**
# Markets	1	1	2
# Symbols	100	100	200
# Strategies	225	250	275
Basis input rate	5,000	5,000	10,000

Clearly, the load level over the SUT is a function of both the input rate and the number of strategies, or algebraically:

$$load = X \cdot W \tag{1}$$

Where X represents the number of events per unit of time and W represents the amount of work required to process each event (which is affected by the number of strategies) – the unit of *load* is therefore expressed as work per unit of time (e.g., work/sec).

However, doubling the input rate, as when going from a scale factor of 10 to 11, does not mean that the system is twice more loaded because a great part of the incoming ticks is matched with fewer strategies. For instance, for `SF=11`, half of the ticks are simply ignored as they are not correlated, one quarter of them are matched with 10 strategies over the first market, and the other quarter is matched with only one strategy over the second market. So, even though the input rate doubled, ¾ of the incoming events are actually ignored or have a much shorter processing path. In formula (1), this means that X doubled, but the average value of W decreased substantially.

We account for that reduction, by splitting the processing of every incoming event into two separate phases and assigning weights to them[22]:

$$W = W_{filtering} + 100 \cdot W_{execution} \quad (2)$$

i. filtering (weight: 1)

ii. passing forward in the execution path of the strategies (weight: 100)

In both phases, the amount of work depends on the number of strategies involved:

$$W_{filtering} = S \quad (3)$$

$$W_{execution} = S_{match} \quad (4)$$

Where S is the total number of strategies in execution and S_{match} is the number of strategies that are actually affected by the incoming tick. As mentioned before, S is straightforwardly derived from the scale factor as follows:

$$S = 25 \cdot SF \quad (5)$$

On the other hand, the number of strategies that are actually executed S_{match} depends on the incoming tick. In particular, one of three things can happen:

i. The tick is simply ignored, as it does not belong to any known correlation;

ii. The tick is matched with exactly 10 strategies, if it belongs to one of the first M-1 markets (for M > 1);

iii. The tick is matched with 1 up to 9 strategies, if it belongs to the last market M.

[22] We assign a small weight for the first phase as in essence it involves only String comparison.

The average value of S_{match} is then given by:

$$\sum_{i=1}^{3} p_i \cdot s_i$$

Where p_i is the probability associated with each of the three situations above, and s_i is the number of strategies executed in each case.

Considering that half of the securities are not part of any strategy, independently on the scale factor, and that only the first M-1 markets have exactly ten strategies, we have:

i. $p_1 = 0.5\ ;\ s_1 = 0$

ii. $p_2 = 0.5 \cdot \left(\frac{M-1}{M}\right)\ ;\ s_2 = 10$

iii. $p_3 = 0.5 \cdot \left(\frac{1}{M}\right)\ ;\ s_3 = (SF-1)\ mod\ 10 + 1$

Which gives:

$$S_{match} = 5 \cdot \left(\frac{M-1}{M}\right) + \frac{1}{2M} \cdot \left((SF-1)\ mod\ 10 + 1\right) \tag{6}$$

From formulas (2) to (6), we have:

$$W = \ 25 \cdot SF + 100 \cdot \left(5 \cdot \left(\frac{M-1}{M}\right) + \frac{1}{2M} \cdot \left((SF-1)\ mod\ 10 + 1\right)\right)$$

Simplifying:

$$W = \ 25 \cdot \left(SF + 2 \cdot \frac{10 \cdot (M-1) + (SF-1)\ mod\ 10 + 1}{M}\right)$$

Since the input rate is a linear function of the number of markets ($X = \lambda \cdot M$), from formula (1) we have:

$$load = \lambda \cdot 25 \cdot \left(M \cdot SF + 2 \cdot (10 \cdot (M - 1) + (SF - 1)\ mod\ 10 + 1)\right)$$

Eliminating the constant (25λ), and since $M = \left\lceil \frac{SF}{10} \right\rceil$, we have the final value for the load, expressed as a function of the scale factor:

$$load = SF \cdot \left\lceil \frac{SF}{10} \right\rceil + 2 \cdot \left(10 \cdot \left(\left\lceil \frac{SF}{10} \right\rceil - 1\right) + (SF - 1)\ mod\ 10 + 1\right) work/sec$$

The metric of *Pairs* is then expressed in terms of the scale factor as:

$$p_{score} = \frac{SF \cdot \left\lceil \frac{SF}{10} \right\rceil + 2 \cdot \left(10 \cdot \left(\left\lceil \frac{SF}{10} \right\rceil - 1\right) + (SF - 1)\ mod\ 10 + 1\right)}{99^{th}\ latency}\ work/sec^2$$

Appendix B

Pairs Benchmark Tools

In this section we describe the tools required for running the *Pairs* benchmark.

B.1 Data Generator

The *Data Generator* application can be executed in either console or graphical mode. In the console mode, the user specifies a couple of parameters in a configuration file and then executes the tool passing that file as argument. The data generation process then starts immediately. If no configuration file is specified, the data generator starts in graphical mode as illustrated in Figure B.1. As it can be seen, when executed in graphical mode, the data generator allows customizing a number of workload parameters, including:

- *Number of symbols* N (default: $100 \times \left\lceil \frac{SF}{10} \right\rceil$)
- *Number of correlations* K (default: 25% of the number of symbols, i.e., half of all the symbols are liable to be monitored by a strategy)
- *Number of strategies* (default: $25 \times SF$)
- *Input rate*:
  - Basis event input rate (default: $5000 \times \left\lceil \frac{SF}{10} \right\rceil$)

- Whether inter-arrival times are exponentially distributed (default) or constant;
- Whether input rate varies over time (default) or not;
- Peak event input rate (default: $1.5 \times basis\ rate$)
- *Test duration* (default: 2 hours)

The output of the data generator tool consists in one or more benchmark input data files containing ticks data, and an auxiliary file describing the number of strategies and the list of correlations for the chosen configuration, which will then be used as input by the query generation tool.

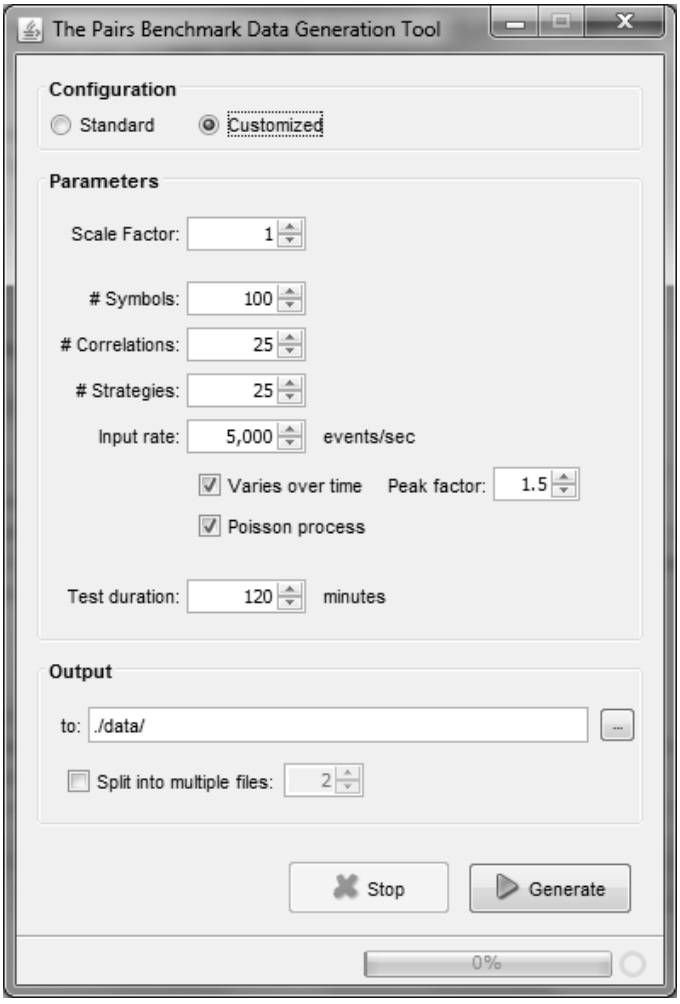


Figure B.1: The *Data Generator* tool (graphical mode).

B.2 Query Generator

The purpose of the *Query Generator* is to produce a neutral representation of the *Pairs* workload that will be later translated into a vendor-specific implementation for the target EP engine. The output of the tool consists in an xml file containing the parameters of the strategies to be executed by the SUT during the benchmark run, as illustrated in Figure B.2. The number of such strategies varies according to the scale factor, as specified in section 7.3.5, and the parameters of each strategy are generated as described in Table B.1 below.

Table B.1: Parameters of a *Pairs* strategy.

Attribute	**Description**	**Possible Values**
alias	Unique identifier of the strategy	Incremental: *st-00001*, *st-00002*, …, *st-0000N*
availableFunds	Amount of funds available for the strategy.	{10k, 20k, 50k}
symbol1	The first security of the strategy.	One of the symbols in the list of correlations produced by the data generator.
symbol2	The second security of the strategy.	
periodLength	The size of the time-based window used in the initial price aggregation (in seconds).	{5, 10, 20, 30, 60}
numPeriods	The size of the count-based window over which the average and standard deviation of ratio are computed.	{5, 6, 7, 8, 9, 10, 15, 20, 25, 30}
bandsMultiplier	The multiplication factor used to compute the upper and lower bands.	{1.2, 1.5, 1.8, 2.0}
stopLossPerc	Threshold used by stop-loss protections.	{10%, 20%, 30%}

```
<Strategies>
 <PairsStrategy
   alias="st_00001"
   availableFunds="1000000"
   symbo11="MCBEIV"
   symbo12="AYFBLW"
   periodLength="20"
   numPeriods="5"
   bandsMultiplier="1.25"
   stopLossPerc="0.3">
 </PairsStrategy>
</Strategies>
```

Figure B.2: Snippet of the output produced by the *Query Generator* tool.

B.3 Translator

The *Translator* is the only part of the *Pairs* benchmark infrastructure that is vendor-specific, which means that a separate translator has to be developed for each target EP engine. Due to the significant differences in the implementation styles adopted by the diverse event processing platforms (see section 2.4), users implementing *Pairs* are free to use any feature or language construct allowed by the target system. The ultimate implementation requirement is to produce the expected answers (i.e., to pass in the validation test). The use of user-defined functions or any other kind of integration with common programming languages is strongly discouraged through, as we consider that any EP system should natively support the set of operations exercised by *Pairs*, and the goal of the benchmark is to assess event processing engines rather than software development environments.

B.4 Test Harness

The *Pairs* benchmark uses the FINCoS framework, introduced in Chapter 3, as its test harness. The task of the framework is to submit load to the system under test and receive answers from it. For that, it reads the data file produced by the *Data Generator* tool, transforms the events on it into an appropriate representation, and send them to the target EP engine. On the opposite direction, FINCoS subscribes to the output streams at the EP engine and stores all incoming results on disk for subsequent validation and performance measurement. The framework is also responsible for assigning timestamps to both input tuples and output results (for the sake of response-time computation).

Note that some previous configuration is required before running performance tests with FINCoS. Detailed instructions on how to use the framework can be found on its user guide [38]. A sample FINCoS test setup file for *Pairs* is also provided in [16].

B.5 Validator

The purpose of the *Validator* application is to verify the correctness of the set of answers produced by the SUT after a benchmark run. For that, it takes the input file created by the data generator and the strategies file produced by the query generator to produce the expected output for this particular configuration. Then, it reads the sink log file, generated by the FINCoS framework, which contains the answers produced by the SUT, and compares it with the expected output.

Like the data generator application, the validation tool can be executed in either console or graphical mode. Once more, to execute in console mode, the application must be executed passing a configuration file as argument. The application will execute and then generate a report like the one shown next.

```
Validation result: FAILED!
 - Indicators:
   # validator answers: 13884
   # SUT answers: 29878
   # correct answers: 13884
   # missing answers: 0
   # undue answers: 15994
   # wrong answers: 0
- Orders:
   # validator answers: 1936
   # SUT answers: 3939
   # correct answers: 1884
   # missing answers: 0
   # undue answers: 2003
   # wrong answers: 52
```

Figure B.3: Output of the *Validator* tool (console mode).

For each output stream, the validator reports:

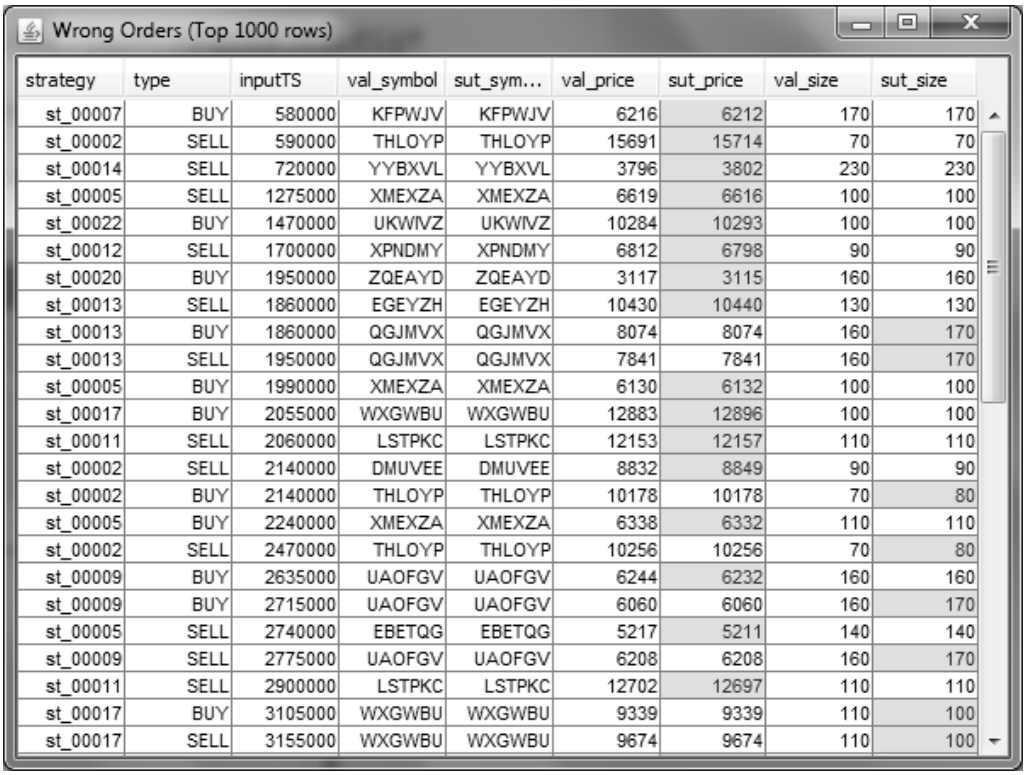
- The number of expected answers (*validator answers*);
- The number of answers produced by the SUT (*SUT answers*);
- The number of correct answers;
- The number of answers that have been generated by the validator, but not by the SUT (*missing answers*);
- The number of answers that have been generated by the SUT, but are not part of the set of expected answers generated by the validator (*undue answers*) and
- The number of answers that were generated by both the SUT and the validator, but with different values (*wrong answers*).

When validation fails, like in the sample report above, the graphical mode of the validation tool (Figure B.4) shall be useful to identify the cause, as it allows visualizing the set of incorrect answers (e.g., see Figure B.5). In the graphical mode, the user specifies the paths for the files containing: (i) the answers produced by the SUT, (ii) the

benchmark input data, and (iii) the parameters of the strategies. In alternative to the last two, it is possible to reuse the answers produced during the last validation ("*use existing*" option) to skip the computation of the expected output – this shall make validation to finish much quicker, but only applies if neither the input data nor the strategies file has changed since last run.

Figure B.4: The *Validator* tool (graphical mode).

After the validation has finished, a report similar to the one of the console mode is displayed, now with the option to visualize the entries, by clicking in the icon on the right side of the results. A window like the one shown in Figure B.5 will then appear.



strategy	type	inputTS	val_symbol	sut_sym...	val_price	sut_price	val_size	sut_size
st_00007	BUY	580000	KFPWJV	KFPWJV	6216	6212	170	170
st_00002	SELL	590000	THLOYP	THLOYP	15691	15714	70	70
st_00014	SELL	720000	YYBXVL	YYBXVL	3796	3802	230	230
st_00005	SELL	1275000	XMEXZA	XMEXZA	6619	6616	100	100
st_00022	BUY	1470000	UKWIVZ	UKWIVZ	10284	10293	100	100
st_00012	SELL	1700000	XPNDMY	XPNDMY	6812	6798	90	90
st_00020	BUY	1950000	ZQEAYD	ZQEAYD	3117	3115	160	160
st_00013	SELL	1860000	EGEYZH	EGEYZH	10430	10440	130	130
st_00013	BUY	1860000	QGJMVX	QGJMVX	8074	8074	160	170
st_00013	SELL	1950000	QGJMVX	QGJMVX	7841	7841	160	170
st_00005	BUY	1990000	XMEXZA	XMEXZA	6130	6132	100	100
st_00017	BUY	2055000	WXGWBU	WXGWBU	12883	12896	100	100
st_00011	SELL	2060000	LSTPKC	LSTPKC	12153	12157	110	110
st_00002	SELL	2140000	DMUVEE	DMUVEE	8832	8849	90	90
st_00002	BUY	2140000	THLOYP	THLOYP	10178	10178	70	80
st_00005	BUY	2240000	XMEXZA	XMEXZA	6338	6332	110	110
st_00002	SELL	2470000	THLOYP	THLOYP	10256	10256	70	80
st_00009	BUY	2635000	UAOFGV	UAOFGV	6244	6232	160	160
st_00009	BUY	2715000	UAOFGV	UAOFGV	6060	6060	160	170
st_00005	SELL	2740000	EBETQG	EBETQG	5217	5211	140	140
st_00009	SELL	2775000	UAOFGV	UAOFGV	6208	6208	160	170
st_00011	SELL	2900000	LSTPKC	LSTPKC	12702	12697	110	110
st_00017	BUY	3105000	WXGWBU	WXGWBU	9339	9339	110	100
st_00017	SELL	3155000	WXGWBU	WXGWBU	9674	9674	110	100

Figure B.5: Viewing the incorrect answers using the *Validator* tool.

B.6 Configuration File

As mentioned in previous sections, a configuration file is required in order to execute most *Pairs* benchmark tools (or at least execute them in headless mode). A sample configuration file is included in the tools package. The file consists in a set of properties as shown below:

```
# Folder where the Data Generator tool saves its output.
dGenFolder=./data/

# Folder where the Query Generator tool saves its output.
qGenFolder=./queries/

# Folder where the Translator tool saves its output.
translatorFolder=./impl/Esper/

# Folder where the Validator tool saves its output.
validFolder=./valid/

# [Data Generator] Benchmark scale factor.
scaleFactor=2

# [Data Generator] Split the data generated by the Datagen tool into one or
more files.
fileCount=1

# [Validator] The file containing the answers produced by the SUT
sutFile=C:\\FINCoS\\log\\SF=2.log

# [Validator] Indicates whether the input file must be processed (set to
false) or not (set to true)
skipInputValidation=true
```

The first four properties indicate where each of the four *Pairs* tools will save its output. The *scaleFactor* property is used by the *data generator* to create input data under the corresponding scale. The *fileCount* property can be used to tell the *data generator* to split its generated data into a given number of files (for instance, for distributing load generation among multiple *drivers*). The *sutFile* property is used by the *validator* tool and indicates the path for the file containing the answers produced by the SUT during the benchmark run. Finally, the *skipInputValidation* property, also used by the *validator*, allows to skip processing of input data to produce the expected answers, by reusing the last validation result.