Pairs Benchmark

Revision 0.2

November 2012

CISUC, University of Coimbra.

Table of Contents

Preamble

This document introduces *Pairs*, a benchmark for evaluating the performance and scalability of event processing platforms[1]. *Pairs* exercises a wide range of features and characteristics commonly found in most event processing applications, including:

- Filtering, aggregation, and correlation of events;
- Detection of event patterns and trends;
- State maintenance, updated upon the occurrence of events of interest;
- Large number of simultaneous queries (increasing with the system scale);
- Changing load conditions.

The benchmark was designed as to assess the ability of the systems in processing increasingly larger number of queries and input rates while providing quick answers – three quality attributes equally important for an event processing engine. *Pairs* was also designed to be fully customizable, so that users can carry out performance studies that resemble more closely their own environments.

In the rest of this document we describe the benchmark workload, metrics and execution rules (Section 1), and introduce the tools available for running it (Section 2).

1 Benchmark Specification

The scenario for *Pairs* is a brokerage house 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 1 below.

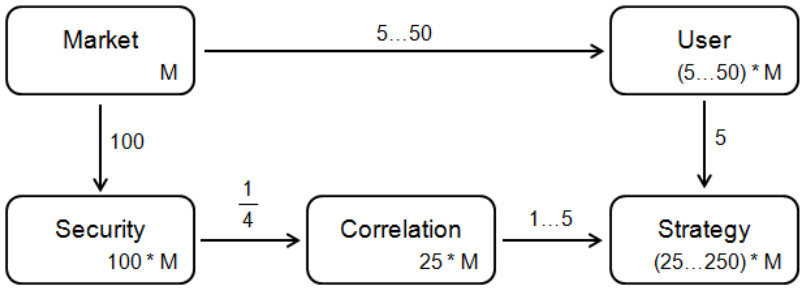


Figure 1: Overview of the benchmark scenario

Per every stock market M, a number of symbols (100) are monitored by the system, from which half are known to be mutually correlated. 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 scale

[1] Throughout this document we use the terms *Event Processing*, *Complex Event Processing* and *CEP* interchangeably to refer to any system capable of processing continuously arriving events with the purpose of identifying situations of interest and reacting upon them.

factor. In the simplest case (5 users), there will be 25 strategies, each defined over a unique pair of correlated symbols. On the limit, each pair of correlated securities is monitored by ten strategies of different users, each with its own parameters. This structure allows evaluating not only if the tested system performs well on a multi-query scenario, but also its ability in sharing resources among similar queries. More details on the scaling of the benchmark are discussed in Section 1.3.

1.1 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**)

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 amount 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)[2].

In the standard configuration, 2 hours of simulated market data is generated by a *data generator* application and submitted afterwards via the FINCoS framework [4] 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 [1][7]. 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 [1], with its λ parameter – which represents the average arrival rate – varying over time, resulting in an arrival pattern as the one illustrated in Figure 2.

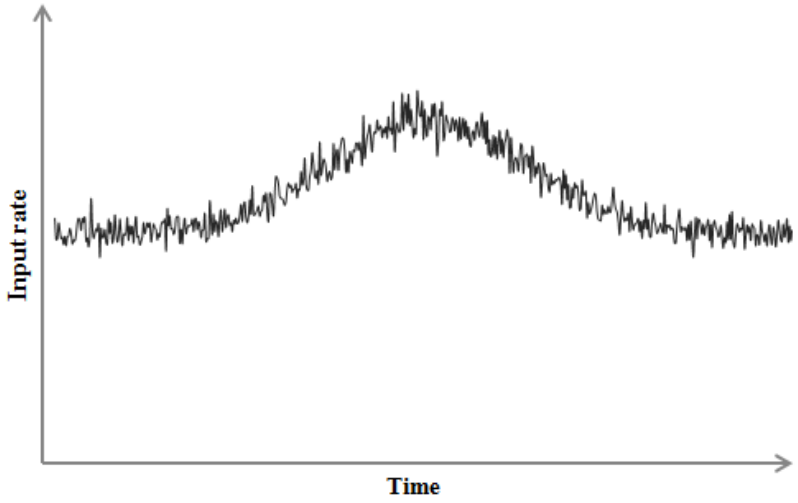


Figure 2: Input rate over time

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

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 how the system responds to progressively larger loads.

1.2 Workload

The benchmark workload consists in processing simultaneously a number of *Pairs* strategies. All strategies perform the same set of operations, described below, although using different parameters:

1. *Compute KPIs*: a couple of metrics are computed to indicate the current state of correlation between the prices of the two monitored securities.
2. *Signal opportunities*: detects when the lines formed by the indicators computed in previous step cross.
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.

The steps above are discussed in further detail on next sections.

1.2.1 KPI Computation

The KPI computation 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 predefined time interval (e.g., the average price during the last 10 seconds). These aggregates are then correlated to produce a *ratio*. Again, the last values of this ratio are aggregated, 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 KPIs is illustrated in Figure 3.

1.2.2 Opportunity Signaling

The three values produced in the previous step are used to determine possible opportunities to capitalize. This happens when the line formed by the values of the ratio crosses either the lower or the upper band, 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(t)*, *Upper(t)*, and *Lower(t)* correspond respectively to the values of the ratio between the securities, the upper band and the lower band at the time instant *t*.

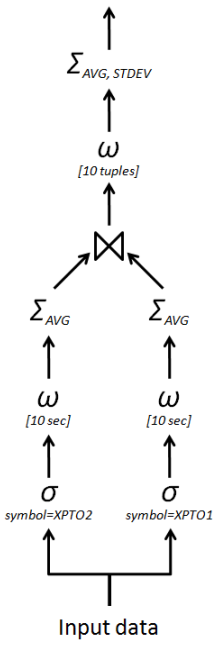


Figure 3: KPI computation

1.2.3 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 4 below illustrates the transitions between the states and their corresponding triggers.

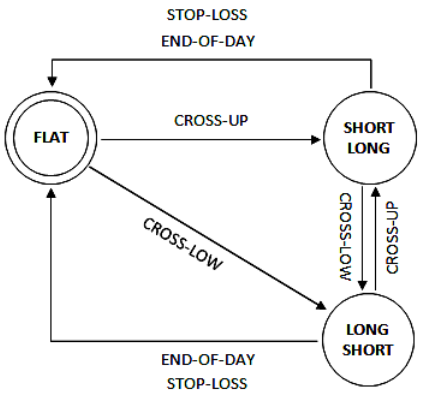


Figure 4: State machine of a *Pairs* strategy

1.2.4 Order Placement

A transition from one state to another is completed only after the corresponding *BUY* and *SELL* orders have been emitted[3]. 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

[3] If the strategy is currently on the FLAT state, only a *buy* order for one of the securities is issued.

the market, for the securities in question. In practice, this means that the system must keep 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 1.

Procedure 1 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 ← GETLAST(toSell);
2:  sellSize ← GETSHARES(toSell);
3:  sellGain ← sellSize * sellPrice;
4:  available = CURRENTBALANCE() + sellGain;
5:  buyPrice ← GETLAST(toBuy);
6:  buySize ← 10 * TRUNCATE(available / (buyPrice*10));
7:  SELL(toSell, sellPrice, sellSize);
8:  BUY(toBuy, buyPrice, buySize);
9:  UPDATEBALANCE(available – buySize);
```

For simplicity, the orders are assumed to be always accepted by the market and executed immediately.

1.2.5 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.

1.3 Output

The output of the *Pairs* benchmark consists in two streams: `Indicator` and `MarketOrder`. 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 5 that allows users to better understand the decisions taken by the strategies*).

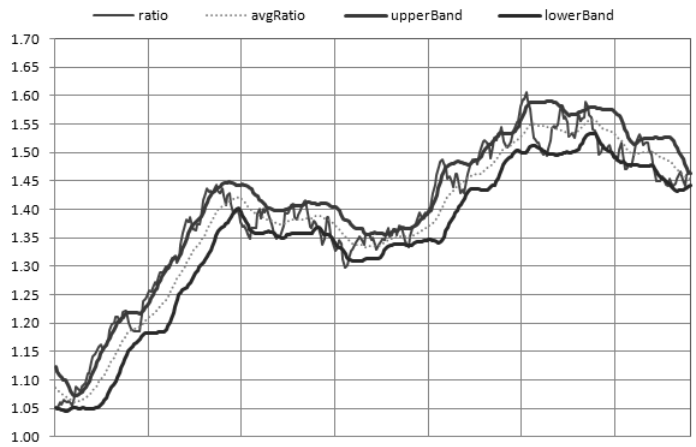


Figure 5: Example of a graph generated with the values of the *Indicator* output stream

The second stream represents the orders that were issued as a result of the execution of each strategy. The schemas of the two output streams are shown below:

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**
)		)	

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 in section 1.2.1. 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 section 1.2.4.

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.

1.4 Scaling

The scale factor (SF) of the benchmark affects the number of users, and consequently the number of strategies executed in parallel as follows:

- Number of users: 5 SF
- Total number of strategies: 25 SF

Additionally, per every increment of ten in the scale factor, the 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 observe 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.

1.5 Measures

The performance of an event processing engine after a run of the *Pairs* benchmark is summarized using the following formula[4]:

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

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 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.

1.6 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*[5]. 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.

[4] An explanation of the term "*load*" and the rationale behind the metric can be found in Appendix A.
[5] 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.

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 6 below:

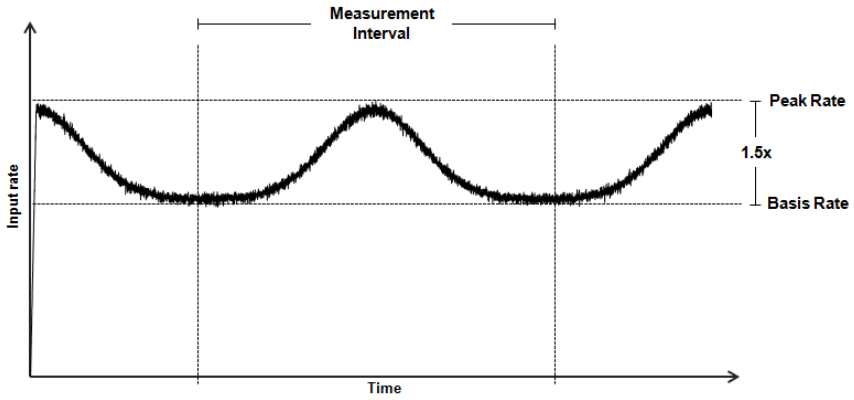


Figure 6: A *Pairs* benchmark run

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 is very likely that the conditions change during their execution. Gracefully responding to these load variations is therefore a fundamental quality that CEP engines should possess.

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). 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.

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.

1.7 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 CEP platforms. First, the workload exercises several common features that appear repeatedly in most event processing applications: it *filters* out ticks from securities which are not of interest, *aggregates* events data over *temporal* and *count-based windows*, *correlates* price data for interrelated securities, *detects patterns* from price movements, keeps track and updates strategies' *state* upon the occurrence of certain events, and performs *lookups* to determine orders price and size. In addition, different from most benchmarks, which have a fixed set of queries (e.g., [2]), 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 CEP 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.

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.

2 Implementation and Tools

The *Pairs* benchmark should be implemented and executed as illustrated in Figure 7 below:

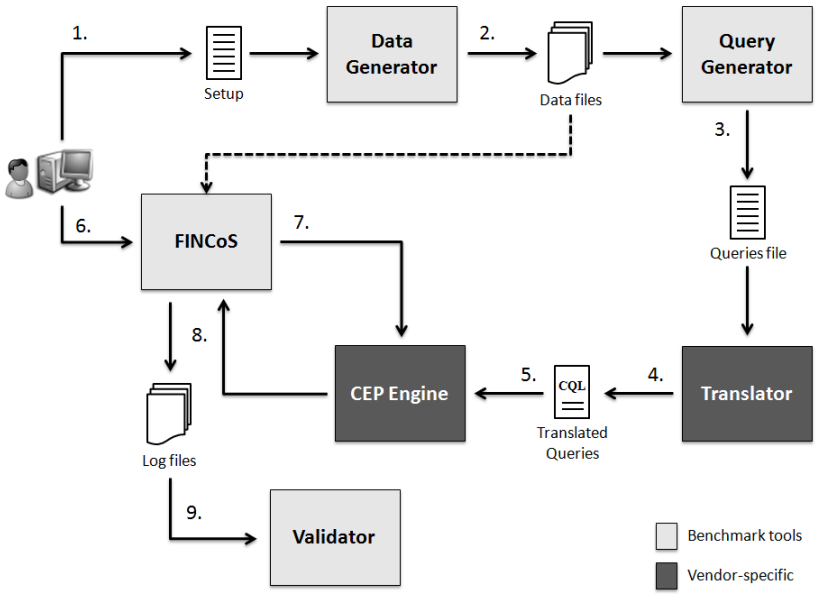


Figure 7: Benchmark execution flow

1. User specifies workload parameters (or uses standard benchmark configuration);
2. A *data generator* application generates one or more data files and auxiliary files to be used by the query generator;
3. A *query generator* creates the strategies into a neutral representation (e.g. 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. Queries/rules are loaded into the SUT;
6. User starts performance run;
7. FINCoS loads generated data file(s) and submits it to the SUT;
8. SUT delivers results to FINCoS;
9. A *validator* verifies the correctness of the answers produced by the SUT

Note that this benchmark infrastructure also allows running tests with real stock market data. For that, the user is required to inform the path for the input data and the list of known correlations between the symbols in the data file (in essence, this means that the set of files provided by the user must be the same as the ones generated by the *data generator*).

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 can be downloaded from [3]. The FINCoS framework is benchmark-

independent and can be found at [4]. In the next sections we describe each tool and provide instructions on how to use them.

2.1 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 it passing the configuration file as argument:

```
java -jar "Data Generator.jar" "Pairs.config"
```

The data generation process starts right away. If no configuration file is specified, the data generator starts in graphical mode:

```
java -jar "Data Generator.jar"
```

A form like the one depicted in Figure 8 will then appear.

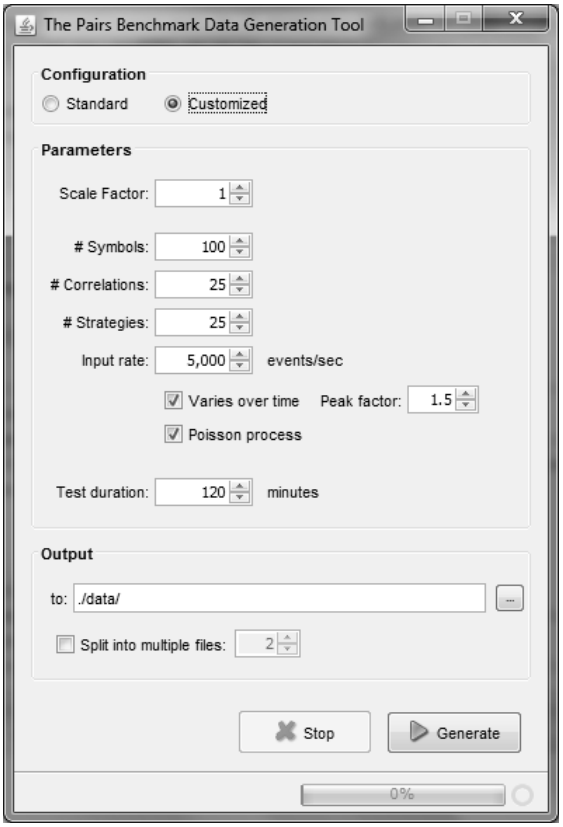


Figure 8: The *Data Generator* tool (graphical mode).

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)

To start data generation, the user clicks the "*Generate*" button. After executing, the data generator produces the data file(s) and a file describing the number of strategies and the list of correlations, which will then be used as input by the query generation tool.

2.2 Query Generator

The function 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 CEP 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 9:

```
<Strategies>
 <PairsStrategy
   alias="st_00001"
   availableFunds="1000000"
   symbol1="MCBEIV"
   symbol2="AYFBLW"
   periodLength="20"
   numPeriods="5"
   bandsMultiplier="1.25"
   stopLossPerc="0.3">
 </PairsStrategy>
</Strategies>
```

Figure 9: Snippet of the output produced by the *Query Generator* tool.

The number of such strategies varies according to the scale factor, as specified in section 1.4, and the parameters of each strategy are generated as follows:

- *alias*: incrementally ("st_00001", "st_00002", etc.);
- *availableFunds*: fixed;
- *symbol11 and symbol2*: iterating over the list of correlations produced by the data generator;
- *periodLength*: takes with equal likelihood one of the values {5, 10, 20, 30, 60};
- *numPeriods*: takes with equal likelihood one of the values {5, 6, 7, 8, 9, 10, 15, 20, 25, 30};
- *bandsMultiplier*: fixed;
- *stopLossPerc*: fixed;

2.3 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 CEP engine. Note that currently there is no standard way to express event processing logic: some products use composition of graphical operators, others production rules, and some others offer continuous query languages. Even in the case of the last two approaches, there is no agreed common syntax or semantics.

For this reason, 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). We strongly discourage though the use of user-defined functions or any other kind of integration with common programming languages, as we believe that any CEP 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 platforms.

2.4 The FINCoS Framework

The FINCoS framework [4] is a set of benchmarking tools for load generation and performance measuring of event processing platforms. The framework was designed to be portable across different CEP products and test scenarios, so it can reused in different benchmarks and/or performance studies.

In the case of *Pairs*, FINCoS is used to submit load to the system under test and receive the answers from it. For that, the framework 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 CEP engine. On the opposite direction, FINCoS subscribes to the results at the CEP engine, and when new tuples arrives, it stores them on disk for later validation.

Note that some previous configuration is required before running performance tests with FINCoS. Detailed instructions on how to use the framework can be found at [4]. Also, a sample test setup file for *Pairs* is provided in [3].

2.5 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 *Validator* can be executed in either console or graphical mode. Again, to execute in console mode, start the application passing a configuration file as argument:

```
java -jar "Data Generator.jar" "Pairs.config"
```

The application will execute and then generate a report like the one shown in Figure 10. 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*).

```
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 10: Output of the *Validator* tool (console mode).

When validation fails (like in the sample report above), the graphical mode will probably be useful to identify the cause, as it allows visualizing the set of incorrect answers (see Figure 11 and Figure 12). To execute in graphical mode, run the same command as before, but without any argument:

```
java -jar "Validator.jar"
```

The following screen will then show up:

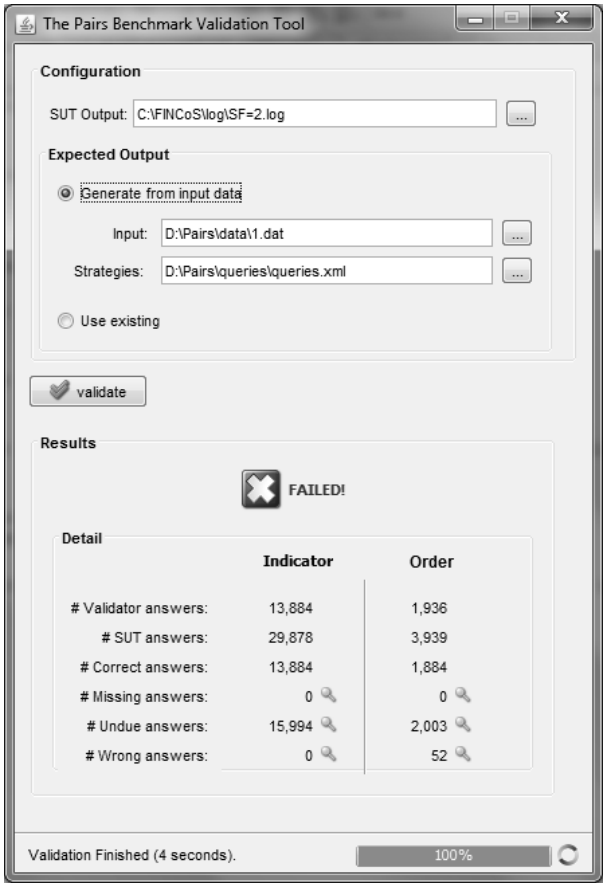
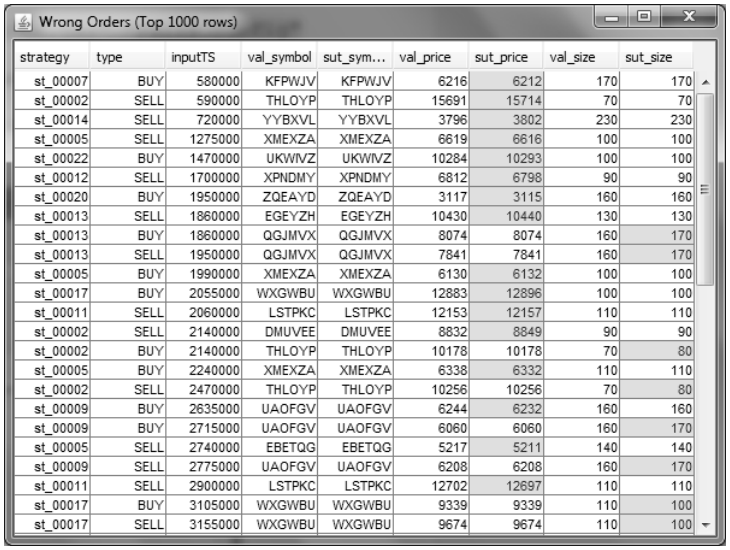


Figure 11: The *Validator* tool (graphical mode).

Again, the user specifies the paths for the files containing: the answers produced by the SUT, the benchmark input data, and the parameters of the strategies. In alternative to the last two, he/she can use 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.

Validation starts by clicking the "*validate*" button. After 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 12 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 12: Visualizing the wrong answers using the *Validator* tool (graphical mode).

2.6 Configuration File

As mentioned before, a configuration file is required in order to execute the *Pairs* benchmark tools. 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.

3 References

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

[2] Arasu, A., Cherniack, M., Galvez, E.F., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., and Tibbetts, R. 2004. Linear Road: A stream data management benchmark. *In Proceedings of the 30th International Conference on Very Large Data Bases* (Toronto, Canada, September 2004), 480-491.

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

[4] FINCoS Framework: `https://code.google.com/p/fincos/`

[5] Mendes, M.R.N., Bizarro, P., Marques, P. 2009. A Performance Study of Event Processing Systems. *In Proceedings of the 1st TPC Technology Conference* (Lyon, France, August 2009), 221-236.

[6] Sachs, K., Kounev, S., Bacon, J., Buchmann, A.P. 2007. Workload Characterization of the SPECjms2007 Benchmark. *In Proceedings of the 4th European Performance Engineering Workshop*, (Berlin, Germany, September 2007), 228-244.

[7] White, S., Alves, A., Rorke, D. 2008. WebLogic event server: a lightweight, modular application server for event processing. *In Proceedings of the 2nd International Conference on Distributed Event-Based Systems* (Rome, Italy, July 2008), 193-200.

APPENDIX

A. The *Pairs* Metric

In this section we discuss the rationale behind the metric of *Pairs*, p_{score}, whose purpose is to facilitate comparison among different benchmark runs and systems. As mentioned in section 1.5, 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 1 below summarizes the differences in the workload parameters for the aforementioned scale factors:

Table 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).

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[6]:

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

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

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.

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

$$\sum_{i=1}^{3} p_i \cdot s_i \quad (6)$$

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. Specifically:

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:

$$S_{match} = 5 \cdot \left(\frac{M-1}{M}\right) + \frac{1}{2M} \cdot \left((SF - 1)\ mod\ 10 + 1\right) \quad (7)$$

And:

$$W = 25 \cdot SF + 100 \cdot \left(5 \cdot \left(\frac{M-1}{M}\right) + \frac{1}{2M} \cdot \left((SF - 1)\ mod\ 10 + 1\right)\right) \quad (8)$$

Since $X = 5000 \cdot M$:

$$load = 1.25 \cdot 10^5 \cdot \left(M \cdot SF + 2 \cdot (10 \cdot (M - 1) + (SF - 1)\ mod\ 10 + 1)\right) \quad (9)$$

Eliminating the constant, 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:

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

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

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