Event Processing on your own Database

Nikolaus Glombiewski, Bastian Hoßbach,
Andreas Morgen, Franz Ritter, Bernhard Seeger

Department of Mathematics and Computer Science
University of Marburg, Germany
{glombien,bhossbach,morgen,ritterf,seeger}@mathematik.uni-marburg.de

Abstract: Event processing (EP) is widely used for reacting on events in the very moment when they occur. Events that require immediate reactions can be found in various applications, e.g. algorithmic trading, business process monitoring and sensor-based human-computer interaction. For the support of EP in an application, a special-purpose EP system with its own API and query language has to be used. Typically, EP systems as well as their integration in applications are expensive. In this paper, we show how every standard database system can be used as an EP system via JDBC. An experimental evaluation proofs that databases behind JDBC are able to support small and medium-sized EP workloads.

1 Introduction

Today, many existing applications benefit from *event processing* (EP) or are enabled by using EP [Luc01]. Algorithmic trading, business process monitoring and sensor-based human-computer interaction are just a few examples of applications that require or take advantage of EP. In order to support EP in an application, a special-purpose EP system has to be used. After the integration, the application code and the EP system are intermeshed (vendor lock-in). Therefore, replacing the EP system for a different one becomes expensive.

In this paper, we motivate and present the implementation of an EP system purely on top of JDBC that enables every standard database system to provide EP functionality. In contrast to existing approaches (e.g. Oracle CEP [ora13] or StreamInsight [G^+09]), an EP system via JDBC is completely independent of specific database products. Therefore, different databases can be used and easily exchanged behind JDBC without affecting the EP implementation on top. Because databases are already integrated in most applications, our approach leads to low costs.

After an brief overview of EP in Section 2, we motivate our approach in Section 3 and present implementation details in Section 4. The proposed approach leads to two advantages. First, every standard database system can provide EP functionality. Second, the user can easily exchange different database products as EP back-end in Java applications. We show in Section 5 that database systems can support small and medium-sized EP workloads via JDBC. Section 6 concludes the main results of this paper.

2 Event Processing

The continuous analysis of streaming events is called *event processing* (EP). EP allows to react immediately when specific events occur. An event is a pair (p, t) consisting of a payload p that is some information about a real or virtual event and a timestamp t that specifies the instant of time when the event occurred. Typically, EP is used to combine events that occurred in a specific order or within a timeframe. For example, an application running on a touch screen device can wait for the user to put exactly three fingers into the upper right quarter of the screen within two seconds (correlation of events). Or a computer game can detect that a player has picked up some item at place A and brought it to place B before a predefined mission timer elapsed (pattern matching of events). Besides correlation and pattern matching, filtering and aggregating events are other important basic EP operators. These four basic EP operators can be combined arbitrarily to express more complex queries. Because every new input event can produce new results, all queries are performed continuously in an event-driven manner.

The implementation of EP in an application comprises three steps that are the core functionality of every EP provider. First, all event stream producing sources (e.g. sensors) are registered at the EP provider. In general, they have a fixed and structured schema that is send to the EP provider during registration. Second, the EP logic is created in the form of a set of continuous queries. Third, sinks (e.g. alarms) are registered at the output side to consume the results of the continuous queries.

3 Motivation

In this section, we give insights into our motivation that led to the development of an EP system on top of JDBC. Inspired by the database abstraction layers ODBC/JDBC, we developed a generic EP abstraction layer that allows the connectivity to different EP providers as well as the building of distributed and federated EP infrastructures. In contrast to databases, the design of such an interface is more complex for the following reasons. First, there is no standard language for EP, perhaps comparable to SQL for databases. The challenge is to design an interface that is powerful enough, but still allows a bridge to any kind of EP provider. Second, operators like pattern matching are not easy to adapt to the underlying EP providers. Our approach for an EP abstraction layer, entitled *Java Event Processing Connectivity*[1] (JEPC), is currently implemented on top of the EP systems *Esper* [esp13] (open source), *Odysseus* [A^{+}12] (academia) and *webMethods Business Events* [web13] (commercial) as shown in Figure 1. Here in this paper, we put our focus on the implementation of an JEPC-to-JDBC bridge. Such a bridge seems to be very appealing for reasonably small EP applications that do not require installing a special-purpose EP system. However, the JDBC bridge can also be used stand-alone to enable every standard database to provide EP functionality. In the following, the most important aspects of JEPC are introduced, because it is used as user interface for our EP system on top of JDBC.

[1]`http://dbs.mathematik.uni-marburg.de/research/projects/jepc/`

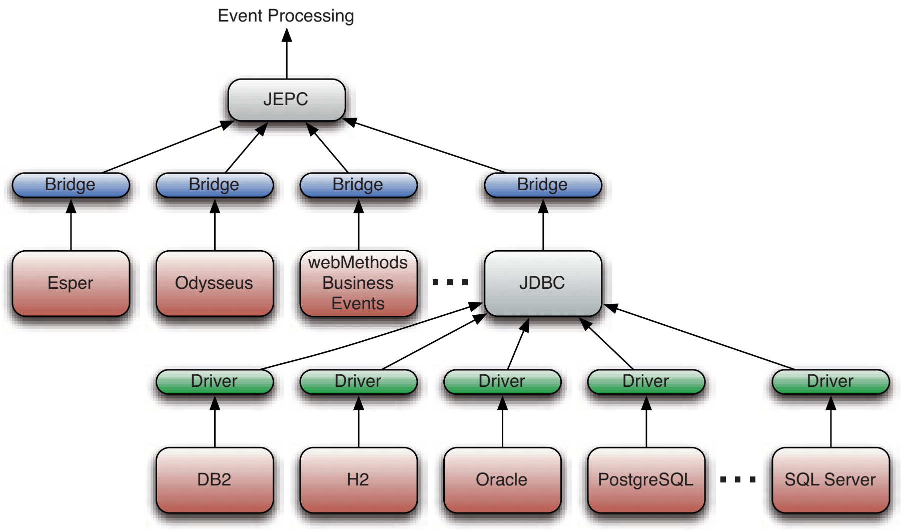


Figure 1: Java Event Processing Connectivity

To abstract EP functionality from specific EP providers, we have designed the JEPC interface shown in Table 1. However, only a unified API is not sufficient. We also have to abstract from the query language, the representation of streams and how query results are made available.

The method to register a new event stream requires a unique identifier e for later accesses and a schema definition s of the stream. Because we want the interface to be generally applicable but also simple, we decided to support the relational model. Thus, a schema is an array of attributes; each attribute consists of a name and a primitive data type.

In the following, we show how to express continuous queries in JEPC. A new query is created by giving it a unique identifier q and a definition d. The query definition makes use of a simple object representation of streams, four basic EP operators and their parameters. This representation has several advantages. First, queries can be composed arbitrarily to build more complex ones. Second, queries can be processed easily. This is very important when an implementation of the JEPC interface (like the bridges in Figure 1) has to translate the input query definition into the specific query language of the underlying EP provider with preservation of semantics. Third, a query definition of JEPC can also be generated easily. This allows the use of query languages, domain specific languages, integrated languages (e.g. LINQ [Mei11] in .NET) and graphical query composers on top of the JEPC interface.

In JEPC, query definitions follow the paradigm everything-is-an-object. A stream is represented by an object that contains all necessary information about the stream (e.g. its name). Each stream object must be related to a parameter object of the type time window [ABW06]. This object holds all information about the time window (e.g. its size). The next group of objects consists of the basic EP operators, namely the filter, the aggregator,

Method	Description
REGISTERSTREAM(e, s)	Registers a new event stream e with schema s
CREATEQUERY(q, d)	Creates a new continuous query q with definition d
ADDOUTPUTPROCESSOR(q, o)	Adds an output processor o to query q
PUSHEVENT(e, p, t)	Pushes the event (p,t) into the event stream e
REMOVESTREAM(e)	Removes the stream e
REMOVEQUERY(q)	Removes the query q
REMOVEOUTPUTPROCESSOR(o)	Removes the output processor o

Table 1: JEPC Interface

the correlator and the pattern matcher. Each of them is related to either one input object (filter, aggregator and pattern matcher) or multiple input objects (correlator). An input object can be a stream object as well as an operator object. On the output side, there is exactly one related output object. This can be another operator object or the final query output object that exists exactly once in every query definition. Each type of operator object needs its own specific parameter object. The filter and the correlator need a boolean expression that specifies all events (filter) respectively combinations of events (correlator), which should be forwarded to the output object. The aggregator needs an aggregate in order to compute its output events and the pattern matcher needs a pattern. Each element of a parameter object (e.g. literals and logical connectors in boolean expressions) is an object again. For the sake of simplicity, we do not mention them in detail here. Because everything is an object, entire parameters and query definitions are formed by composing objects. The final composition determines precisely all relations between the objects and makes it easy to traverse and translate the query definition as well as to create and modify it. In other words, queries are expressed in the form of a composition of primitive operators (directed EP operator graphs). Then, every bridge only needs a compiler that is able to translate the basic EP operators into the specific query language of the underlying EP provider with preservation of semantics. Following the well-known concept of subqueries, complex EP queries can be translated and executed as event processing networks [Luc01].

The semantics of JEPC queries is the same as the one presented in [KS09]. We have decided to adapt this one, because it is sufficiently expressive, deterministic, consistent and has a solid theoretical foundation (at each single point in time the queries produce the same results as standard SQL would do). Pattern matching is performed strictly sequential with time instant semantics [WDR06]. Each bridge guarantees to preserve the defined semantics.

To consume the results of a continuous query q, the user can add multiple output processors. Inside an output processor o, user-defined code is executed on each produced result. A new event (p, t) is pushed into a registered stream e by calling the corresponding method of the JEPC interface. Additionally, there are also methods needed to remove streams, queries and output processors.

4 JEPC-to-JDBC Bridge

In this section, we present our data structures and algorithms being used on top of JDBC to implement the JEPC interface. Time windows are the most important data structure of EP. They are used as building blocks to correlate and aggregate events. Pattern matching is the most challenging operator. For the sake of limited space, we assume that every input stream is ordered by time. But available techniques to handle out-of-order streams (e.g. on the basis of punctuations [TM03]) are fully compatible with our approach.

4.1 Query Translation

The JEPC-to-JDBC bridge translates each incoming JEPC query definition into standard SQL with additional clauses for time windows and pattern matching. See Listing 1 for examples of the syntax we use for the additional clauses. The main idea is to treat the SQL extensions accordingly so that the resulting statement will be pure standard SQL. Finally, the bridge can execute the resulting statements in the database via JDBC. Of course, it is also possible to express EP queries direct in the form of our extended SQL syntax without using JEPC (stand-alone mode).

4.2 Time Windows

Because event streams are potentially unbounded, they can not be queried as a whole. Instead, sliding time windows [ABW06] are used to keep a finite set of the freshest events and to perform the query execution. The sizes of time windows are specified on the source streams by the user. For all other streams obtained from an operator, the sizes of time windows are derived from the specification of the operator and its corresponding input streams. The JEPC-to-JDBC bridge creates for each time window a new database table. In the associated query statements, it replaces each source stream together with its time window specification by the corresponding database table. The resulting expression is already a valid SQL statement being ready for execution in a database system. Figure 2 visualizes this procedure. When a new event is added to an event stream, all dependent tables that represent a time window on it are updated [DR04]. Because the freshest event is the one that was recently added to the window, we can use its timestamp and the size of the corresponding time window to deduce an instant of time as a bound for purging. Events dating back before this bound are expired; therefore, they are deleted. In Figure 2 the update is shown for *table_a*. Afterwards, all tables only contain relevant events and it is possible to execute queries on them.

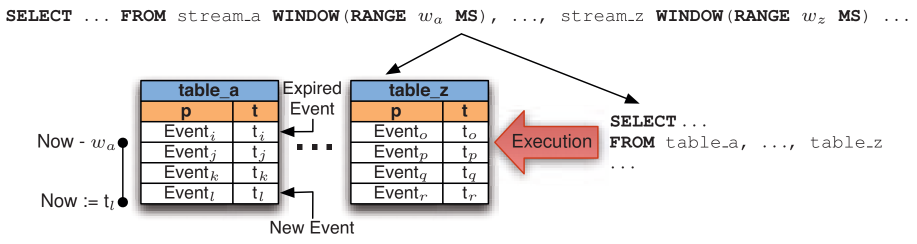


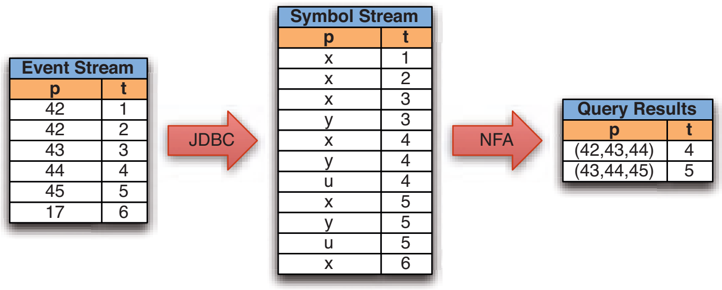
Figure 2: Time Windows

4.3 Filter, Correlation and Aggregation

Although we have stated in the last section that the rewritten queries can be executed on the simultaneously created tables such that all results are produced, we have to perform further modifications in order to achieve the behavior of an EP provider. The reason why we need to tweak the execution lies in the stream-based nature of EP. This especially means that the output of every operator must be an event stream again. So we are forced to report every result exactly once and to report all results ordered globally by time. An operator result is called new if the recently added event took part in its creation. Every EP operator has to report only new results.

To produce only new results, we created an environment capable of recognizing the recently added event. In our implementation, each single stream has an extra table that always contains the freshest event of the stream only. This allows to select it efficiently by querying this extra table instead of a time window. For the rest of the paper, we use the terms table and time window synonymously and call the extra table the *inbox* of a stream. Each stream has exactly one inbox and can have multiple time windows (the total count of time windows depends on the running EP operators).

After the common setup, the filter, correlation and aggregation operators are handled as follows. A filter basically checks whether an incoming event fulfills certain requirements or not. Thus, the filter is executed on each new event in the inbox and returns the result directly. An aggregation can not exploit the inbox, because it has to take every valid event inside a time window into consideration. So the time window on its input stream is updated and the aggregation is executed as described in the previous section. A correlation between event streams requires the inbox of the updated input stream in order to report only new results. The recently added event is inserted into its corresponding time window and expired events are removed in every time window on the input streams of the correlation with the timestamp of the recently added event as upper bound. The recently added event not only let the time progress in its corresponding time window but also in all others. Finally, the inbox that contains only the recently added event is correlated with all other time windows. This produces all results in that the recently added event takes part [BLT86].



Event Stream

p	t
42	1
42	2
43	3
44	4
45	5
17	6

Symbol Stream

p	t
x	1
x	2
x	3
y	3
x	4
y	4
u	4
x	5
y	5
u	5
x	6

Query Results

p	t
(42,43,44)	4
(43,44,45)	5

Figure 3: Pattern Matching Example

4.4 Sequential Pattern Matching

Creating a new pattern matching query via JEPC requires the following information in the query definition: a strictly sequential pattern that is described through a regular expression, boolean expressions that specify the mapping between events of the queried stream and the symbols used in the pattern, a time window within the pattern must occur completely, arbitrary global variables to hold values globally for the pattern and an output event consisting of a subset of the global variables. The test query ρ in Listing 1 shows the resulting SQL statement after the bridge has translated the incoming query definition. For each parameter there is an additional clause: the pattern-clause contains the regular expression, the within-clause contains the duration of the time window and the measures-clause declares all global variables. Inside the define-clause, each symbol is defined through a boolean expression (as-clause). This regular expressions can use constants, attributes of the event stream as well as previously set global variables as literals. The do-clause is used to set global variables when the corresponding symbol is emitted (that is the boolean expression was evaluated to true). Figure 3 shows the test query ρ in execution with i set to 1. In this case, the query detects all integer event sequences of size three. Furthermore, each integer event in a matching sequence must have a value that is decreased exactly by one in comparison to the value of its successor.

For each new event in the inbox of the stream, all symbols it emits are derived. Because symbols are defined through boolean expressions, this step is done completely inside the database. The result is a symbol stream that is maintained in the same way as every other event stream. Especially the specified time window is used to hold only relevant symbols. Also the values of global variables are derived and stored inside the database. Because in strictly sequential pattern matching all symbols with identical timestamps are interpreted as alternatives, we manage all alternative sequences contained in the symbol stream inside the database and give each of them a unique identifier. The same identifiers are used to tag the corresponding values of the global variables. All symbols and the identifiers are put into a *nondeterministic finite automaton* (NFA) that is built on the basis of the pattern. This NFA reports all identifiers of sequences that have a positive match with the pattern. Finally, the identifiers are used to load the corresponding values of the global variables in order to build and report the output event.

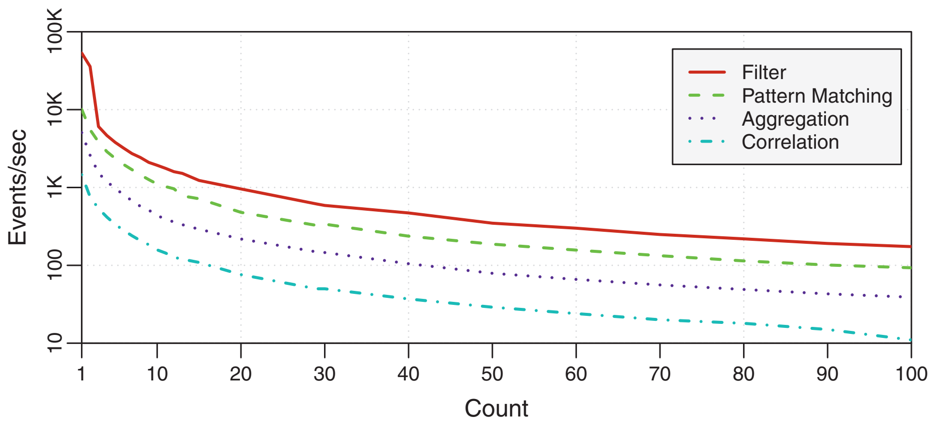


Figure 4: Workload Scalability

5 Evaluation

In order to obtain an impression of the performance of our EP system on top of JDBC, we conducted an experimental evaluation. Two things are remarkable about our implementation and test environment. First, our implementation is strictly single-threaded and consumes only few resources (in the worst case, one CPU core is fully utilized). We avoided to implement a multi-threaded EP server, because our target use-cases are applications that need embedded EP functionality on the same machine. Second, we used the database system *H2* [h2d13] in our experiments, because it is a simple Java library and in the presence of JEPC, users only have to add this library to their projects to enable EP. There is no complicated setup required.

5.1 Experiments

We have examined for each basic EP operator the number of input events being processed per second. We also have evaluated the scalability of all basic EP operators by running multiple of them at the same time. Our test machine had an Intel i7 CPU with 8 GB main memory. The database system *H2* used its standard configuration keeping its data on an ordinary hard disk. Figure 4 shows the results on a logarithmic y-axis.

```
$\sigma(i, Count)$ = SELECT *
               FROM   stream[a:Integer, b:Integer]
               WHERE  a - b = $i$

$\bowtie(i, Count)$ = SELECT *
               FROM   stream1[a:Integer] WINDOW(RANGE $(500 + (Count/2) - i)$ MS),
                      stream2[b:Integer] WINDOW(RANGE $(500 + (Count/2) - i)$ MS)
               WHERE  a - b = $i+1$

$\alpha(i, Count)$ = SELECT COUNT(*)
               FROM   stream[a:Integer] WINDOW(RANGE $(500 + (Count/2) - i)$ MS)

$\rho(i, Count)$ = SELECT z1, z2, z3 FROM stream[a:Integer]
              MATCHING(PATTERN  xyu WITHIN $(500 + (Count/2) - i)$ MS
                       MEASURES z1:Integer, z2:Integer, z3:Integer
                       DEFINE   x AS true       DO z1 = a
                                y AS a - z1 = $i$ DO z2 = a
                                u AS a - z2 = $i$ DO z3 = a)
```

Listing 1: Test Queries - Filter σ, Correlation $\bowtie$, Aggregation α and Pattern Matching ρ

In the experiments, exactly one event was pushed into all input event streams for every millisecond. So all adjacent events in an event stream had timestamps that differed exactly by one millisecond from each other. Therefore, all time windows of the parametrized test queries (see Listing 1) were filled with 500 events on average. Before each measurement, we performed a warm-up. During this phase, all time windows were filled up completely. We executed the following sets q_{Count} of test queries three times and have reported the average input event throughput:

$$\forall q \in \{\sigma, \bowtie, \alpha, \rho\} : \forall Count \in \{1, \dots, 100\} : q_{Count} = \{q(i, Count) \mid 1 \leq i \leq Count\}$$

Independent of specific operator types, the throughput decreases rapidly at the beginning and becomes relative stable afterwards. An extract of the precise results is discussed in the following: 80 parallel running filter operators processed 219 events per second, 80 parallel running pattern matching operators processed 114 events per second, 80 parallel running aggregation operators processed 49 events per second and 80 parallel running correlation operators processed 18 events per second. Aggregations are expensive, because they are computed entirely from the scratch and not incremental like in real EP systems. Performance optimizations like incremental computation of aggregates are generally possible and will be addressed in our future work. We also tested a mixed workload consisting of 20 filter, 20 aggregation, 20 correlation and 20 pattern matching operators and measured an average event throughput of 100 events per second. This performance is sufficient for many EP tasks as well as for the development and testing of EP applications. We executed also all tests with the database on a solid state drive and in-memory, but the performance improvements were only in the range of 1 to 10 percent, so we do not report them in detail.

Our implementation behaves like most special-purpose EP systems; for each single input event all related queries are triggered (tuple-driven behavior). We would achieve better performance if queries are only triggered after some time has elapsed or a predefined number of input events have been pushed (batch-driven behavior).

6 Conclusion

We have motivated and presented the implementation of an *event processing* (EP) system purely on top of JDBC. This kind of implementation enables every standard database system to provide EP functionality in Java applications. To support small and medium-sized EP workloads, a database system that in most applications already exists can be (re-)used with only low costs. The use of a database system instead of a special-purpose EP system makes also sense for the development and testing of EP applications. Because JDBC abstracts from specific database products, they can be easily exchanged without affecting the implementation of the EP system on top of JDBC.

Acknowledgments This work has been supported by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant no. 01BY1206A.

References

[A+12] H. Appelrath et al. Odysseus: a highly customizable framework for creating efficient event stream management systems. In *DEBS*, pages 367–368, 2012.

[ABW06] A. Arasu, S. Babu and J. Widom. The CQL containuous query language: semantic foundations and query execution. In *VLDB Journal*, 15(2), pages 121–142, 2006.

[BLT86] J. Blakeley, P. Larson and F. Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, 1986.

[DR04] L. Ding and E. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM*, pages 98–107, 2004.

[esp13] Esper. `http://esper.codehaus.org/` (2013-01-17).

[G+09] T. Grabs et al. Introducing Microsoft StreamInsight. Technical report, 2009.

[h2d13] H2 Database. `http://www.h2database.com/` (2013-01-17).

[KS09] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. In *TODS*, 34(1), pages 4:1–4:49, 2009.

[Luc01] D. Luckham. The power of events: an introduction to complex event processing in distributed enterprise systems. Addison-Wesley Longman Publishing, 2001.

[Mei11] E. Meijer. The world according to LINQ. In *Commun. of ACM*, 54(10), pages 45-51, 2011.

[ora13] Oracle CEP. `http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html` (2013-01-17).

[TM03] P. Tucker and D. Maier. Dealing with disorder. In *MPDS*, 2003.

[WDR06] E. Wu, Y. Diao and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[web13] webMethods Business Events. `http://www.softwareag.com/corporate/products/wm/events/` (2013-01-17).