Event Processing IN ACTION

SAMPLE CHAPTER

Opher Etzion
Peter Niblett

FOREWORD BY DAVID LUCKHAM

MANNING

Event Processing in Action

by Opher Etzion and Peter Niblett

Chapter 10

brief contents

Part 3

Pragmatics

Finally, the wedding event has been planned and is about to take place. There are pragmatic decisions to be made concerning the duration of the event, weather problems anticipated for the day (which might necessitate a move indoors), table seating arrangements, and more.

In this part of the book we discuss some of the pragmatics behind the implementation of an event processing system. Chapter 10 deals with the engineering aspects. It looks at the various programming languages that are available, and some of the engineering questions involved in implementing the non-functional properties of event processing. Chapter 11 deals with pragmatic challenges involved in implementing event processing applications using the current state of the practice, in areas like getting events in the right order and dealing with inexact events.

Chapter 12 completes the book by discussing current usage trends and looks at emerging directions in event processing.

10 Engineering and implementation considerations

In theory, there is no difference between theory and practice; in practice, there is.

—Chuck Reid

In the second part of this book, we concentrated on the principles of event processing, giving examples to show how these principles are used in practice. In this chapter we change our emphasis and focus explicitly on the implementation of event processing applications. The implementation-related topics we discuss are language styles, non-functional properties, performance optimizations, and validation. These are the major engineering topics behind implementation of event processing applications. In summary, this chapter discusses the following topics:

- Event processing programming in practice surveying various programming styles found in current products
- Non-functional requirements of event processing applications
- Performance metrics for event processing applications
- Optimization techniques in event processing

NOTE We discuss some specific performance optimizations in section 10.3, but before we get there we point out a couple of reasons why an event processing approach can eliminate bottleneck issues that you might encounter if you were to use a centralized server approach, such as loading all the events into a database management system (DBMS) server. The first of these is asynchronous execution, which we mentioned in chapter 1. The

processing of events can be split into a chain of operations (event processing agents in our model), which can execute asynchronously to one another on separate processors. The second reason is that you can often implement a single event processing operation using multiple runtime artifacts executing on different processors.

10.1 Event processing programming in practice

At the time of writing this book, there are no standards for event processing programming languages, although there are various programming styles and approaches. The building block approach that we use in this book is a kind of modeling language and, as you can see from the different code samples, there are various ways to implement each of these building blocks. In this section we survey some of the most common event processing programming styles, both the language itself and the type of development environment used with it.

In this section we look at two styles, which we term the stream-oriented style and the rule-oriented style. We also survey different types of development environments. There is a third style, the imperative style, where the logic is coded in a C- or Java-style language. There are several languages like this, but they vary quite a bit from one another, so we suggest you look at examples of specific languages. The Apama MonitorScript language is a good representative of the imperative style and we showed an example of it in chapter 8.

This section is based on the tutorial made by the Event Processing Technical Society (EPTS) Language Analysis group[1] in July 2009.

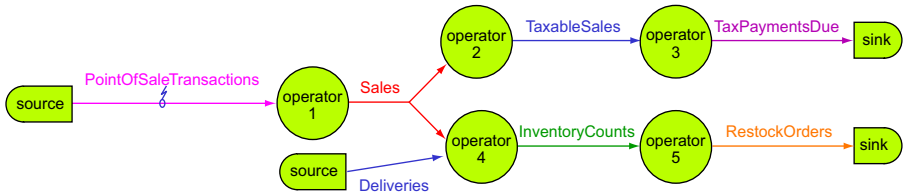
10.1.1 Stream-oriented programming style

The stream-oriented programming style is rooted in data flow programming. In essence a data flow graph is a directed graph that consists of nodes and edges. The nodes represent processing elements, and the edges represent data flowing between these nodes. The paradigm is one of continuous queries, sometimes called *operators*, that are constantly running in the nodes, while their results flow through the edges in the data flow graph. Note that the EPN discussed in this book can be represented as a data flow graph.

The languages used to describe the queries are inspired by SQL and relational algebra, though not all of them are based on SQL. As noted when we discussed stream computing in chapter 2, streams are not necessarily streams of events, and indeed some of the roots of stream programming come from signal processing. When we are using a data flow graph for event processing, the data flowing in the streams are event instances and have the appropriate event semantics.

These event instances are represented as records, and are often referred to as *tuples* following the relational model's terminology. A stream is a continuous flow of events,

[1] The full tutorial made by the EPTS language analysis group in ACM DEBS 2009 is available in http://www.slideshare.net/opher.etzion/debs2009-event-processing-languages-tutorial.




```
stream PointOfSaleTransactions(...) := Source()[...]{...}
stream Sales(...) := operator1(PointOfSaleTransactions)[...]{...}
stream TaxableSales(...) := operator2(Sales)[...]{...}
stream TaxPaymentsDue(...) := operator3(TaxableSales)[...]{...}
Null := Sink(TaxPaymentsDue)[...]{...}
stream Deliveries(...) := Source()[...]{...}
stream InventoryCounts(...) := operator4(Sales;Deliveries)[...]{...}
stream RestockOrders(...) := operator5(InventoryCounts)[...]{...}
Null := Sink(RestockOrders)[...]{...}
```

Figure 10.1 An example of a data flow graph with streams on the edges and operators on the nodes

in most cases all of the same event type, and are considered to be tuples of the same relation. The stream may be unbounded and be active forever. This means that, unlike the conventional relational model where a query is executed against an entire table of data, in the continuous query model a query can execute only against a bounded subset of the stream. The stream is therefore broken up into a sequence of *windows* and the query is performed successively against each window. Windows in stream processing correspond to the temporal context concept that we defined in chapter 7 (and for this reason we sometimes refer to temporal context partitions as windows).

Figure 10.1 shows a data flow graph, in which the edges represent streams, and operations on streams are represented by the nodes. This data flow graph is taken from the SPADE language.[2]

There is another way of representing the graph, shown in figure 10.2, which is taken from the Aleri language. In this representation the nodes represent derived streams (and so incorporate a derivation operation) and the edges show the flow of events by indicating which streams provide input to the derivation operation. In figure 10.2 you can see an example in which the `ValueByBook` stream is derived by an aggregation operation performed on the `IndividualPosition` stream.

You can see from these examples that there are several ways to model stream processing. We now show some examples of stream processing code. Note that these are just samples; to learn the details of a particular language refer to the fuller examples and references on the book's website.

[2] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, Myungcheol Doo: SPADE: the system's declarative stream processing engine. SIGMOD Conference 2008: 1123-1134. http://portal.acm.org/citation.cfm?doid=1376616.1376729.

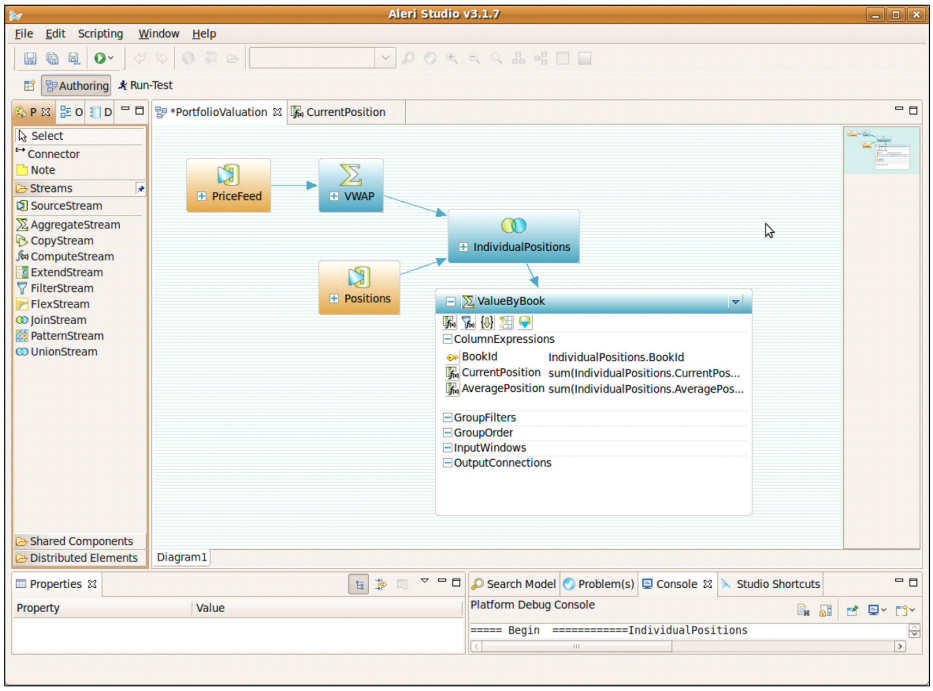


Figure 10.2 An example of data flow with a node as a record and associated operation taken from the Aleri language

Here is an example of a query in the CQL language (developed in the Stanford Stream project).

```
Select Sum(O.cost)
From Orders O, Fulfillments F
[Range 1 Day]
Where F.clerk = "Sue"
   And O.customer = "Joe"
   And O.orderID = F.orderID
```

1 Input streams
2 Window
3 Match condition

This query composes the two input streams shown at 1, applying a temporal context 2 to the second input stream and using the match condition 3. This query adds up the total cost of orders placed each day by customer "Joe" that were fulfilled by clerk "Sue".

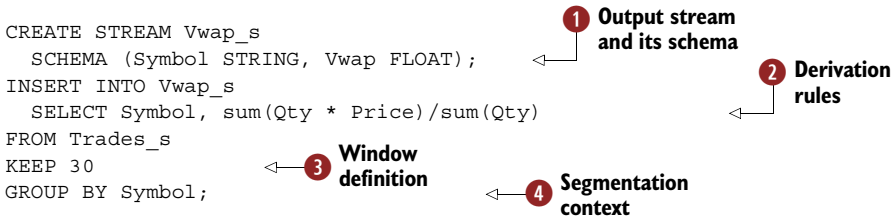
Our next example is a more complex CQL query; this one includes a segmentation context and sampling. Sampling is a form of filtering used in stream computing as the number of events in each window may be high.

```
Select F.clerk, Max(O.cost)
From Orders O, Fulfillments F
[Partition By clerk Rows 5]
10% Sample
Where O.orderID = F.orderID
Group By F.clerk
```

1 Window
2 Sampling directive
3 Segmentation context

This query takes a 10 percent sample of the `Fulfillments` stream, and extracts the five most recent fulfillments for each clerk (this is specified by the window specification ❶ and the sampling directive ❷). As in regular SQL, the `Group By` ❸ means that the query then computes the maximum order cost for each of these groups of five fulfillments. The combination of `Partition By` and `Group By` is equivalent to a segmentation context in our model.

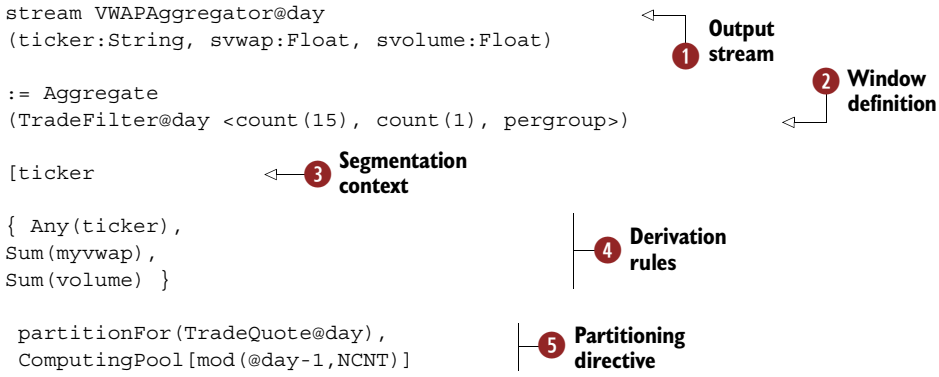
Here's an example of a different language, also SQL-based. This one uses Continuous Computation Language (CCL), the language used by Coral8 (now part of Aleri).



This query takes as an input a stream of `Trade` events, and produces an output stream of derived events ❶ containing volume weighted average prices. The calculation is specified by the derivation rules ❷, which follow standard SQL syntax (as was the case with the CQL examples). There is a 30-minute time window ❸, and the `GROUP BY` clause ❹ establishes a segmentation context meaning that volume-weighted average price (VWAP) calculations are performed independently for every different stock symbol encountered in the input stream.

We end with listing 10.1, which shows an example of an operator written in a stream processing language that does not look like SQL.

Listing 10.1 An example of an operator written in a stream processing language



This is a SPADE aggregate operator, which takes a set of VWAP values as input and then adds them up ❹. As with the CCL example, it starts with a definition of the output stream (in this example it's called `VWAPAggregator@day`) and its schema ❶. The window definition ❷ means that the operator takes input from the `TradeFilter@day` stream, and calculates its aggregate every time a trade occurs, using the last 15 trades

(this is similar to our sliding event temporal context). The operator also includes a segmentation context ❸ that means that the calculation is performed separately (and in parallel) for every distinct value of the `ticker` attribute. The last lines ❺ tell the system where to locate this processing for optimal performance in a multiprocessor system.

To summarize, stream-oriented languages are one of the common event processing language styles. Although several of these languages extend SQL, different languages extend it in different ways.[3] We now look at another style: rule-oriented languages.

10.1.2 Rule-oriented languages

The other dominant style of event processing languages is the style we call *rule-oriented*. The *rules* word is overloaded, as there are several distinct types of rules: production rules, active (event-condition-action) rules, and rules based on logic programming. We briefly survey each of these styles.

PRODUCTION RULES

Production rules are rules of the type *if condition then action*. They operate in a forward chaining way: when the condition is satisfied, the action is performed. Production rules are rooted in expert systems; the operational processing of production rules may be either declarative or procedural:

- Declarative production rule execution is typically based on a variation of the Rete[4] algorithm which matches facts against the patterns contained in the rules to determine which rule conditions are satisfied. Information about the antecedents (conditions) of each rule is stored in an internal state, and in every execution cycle changes to these states are evaluated.
- Procedural production rule execution is based on sequential execution of compiled rules.

Production rules are based on state changes and not on events; however, some event processing languages extend Rete-based production rules to support event processing. This is done by making events an explicit part of the model, so that event occurrences can be used as part of the conditions for invoking an inference rule. Thus the event processing is done through an inference process.

Figure 10.3 shows the Object Management Group (OMG) Production Rule Representation classes. This figure comes from part of an OMG standard for modeling production rules in UML. As noted, events are modeled as part of the rule conditions.

---

[3] This article,presented by owners of different languages, discusses the semantic differences among stream SQL extensions: Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, Stanley B. Zdonik: *Towards a streaming SQL Standard*. PVLDB 1(2): 1379-1390 (2008). http://www.vldb.org/pvldb/1/1454179.pdf.

[4] The Rete algorithm was introduced in Charles Forgy: *Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem*. Artif. Intell. 19(1): 17-37 (1982).

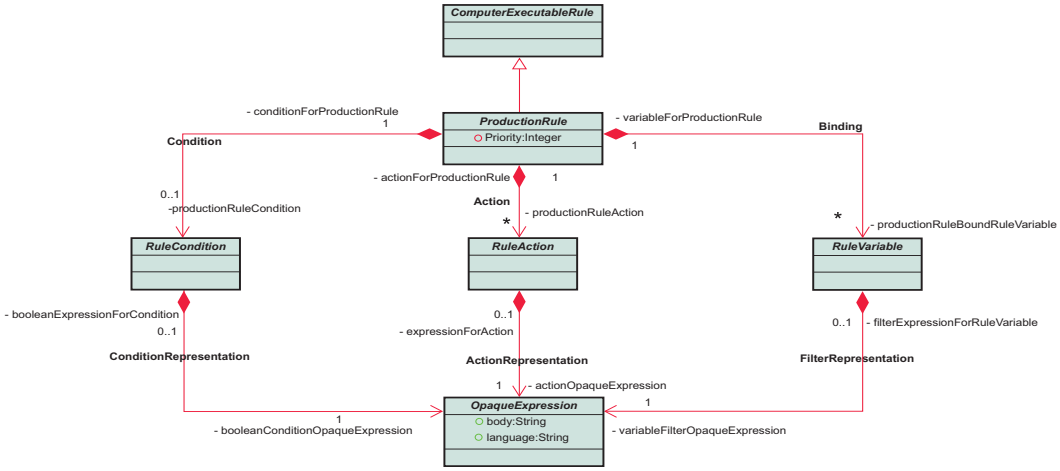


Figure 10.3 OMG Production Rule Representation

ACTIVE RULES

Active rules, also known as event-condition-action (ECA) rules, are descended from work on active databases.[5] Active rules operate according to the following execution pattern: when an event occurs, evaluate conditions and, if they are satisfied, trigger an action.

The event may be primitive or composite. The action can be one that derives an additional event, in which case an active rule maps directly onto an EPA in our model. In cases where the action performs some external activity, such as invoking an external service, the rule maps to the combination of an EPA and an event consumer. In listing 10.2 we show the general structure for active rules.

Listing 10.2 General structure for active rules

```
<Rule style="active" eval="strong">
      <on>
                  <!-- event -->
      </on>

      <if>
                  <!-- condition -->
      </if>

      <do>
                  <!--  action -->
      </do>

     <ifPost>
                  <!-- postcondition -->
      </ifPost>
```

[5] Norman W. Paton, *Active Rules in Database Systems*, Springer, 1998, http://www.amazon.com/Database-Systems-Monographs-Computer-Science/dp/0387985298/ref=sr_1_11?ie=UTF8&s=books&qid=1259266096&sr=8-11.

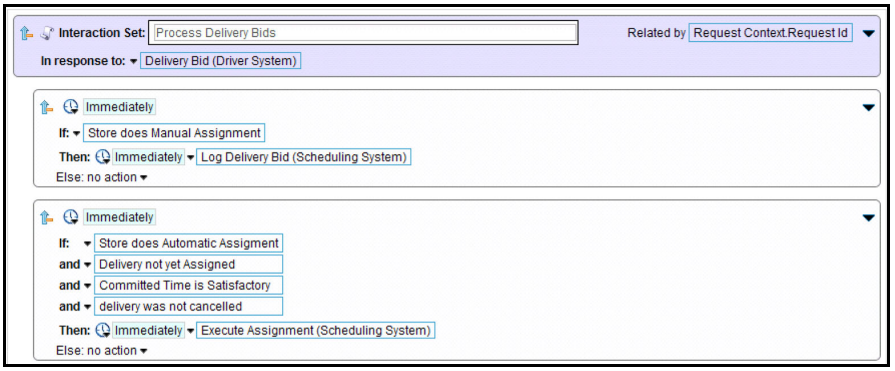


Figure 10.4 An example of an active rule in IBM WebSphere Business Events

```
    <doAlternative>
                    <!-- alternative/else action -->
    </doAlternative>
</Rule>
```

This is a general structure for active rules; particular rule languages are variations of this structure. An example of a particular active rule language is IBM WebSphere Business Events, as illustrated in figure 10.4.

This is an active rule taken from the Fast Flower Delivery application that filters the bids according to the store's preference to perform manual or automatic assignments and routes to the appropriate action.

The third kind of the event processing rule language is the logic programming rule style.

LOGIC PROGRAMMING RULES

Logic programming is a programming style based on logical assertions. The most well-known example of a logic programming language is Prolog. The application of the logic programming style to event processing is rooted in the work done in the deductive database area.[6]

Listing 10.3 shows an example of event processing based on logic programming, taken from the ETALIS implementation of the Fast Flowers Delivery application. Refer to the *Event Processing in Action* website for more information on the syntax and semantics of the language.

Listing 10.3 Example based on ETALIS logic programming

```
no_bid_alert(DeliveryRequestId)<-
        start_automaticAssignment(
            DeliveryRequestId,
            StoreId,
```

1 Automatic assignment

[6] A good source of knowledge about deductive databases is Stefano Ceri, Georg Gottlob, Letizia Tanca, *Logic Programming and Databases*, Springer-Verlag, 1990. http://www.amazon.com/Programming-Databases-Surveys-Computer-Science/dp/0387517286/ref=sr_1_7?ie=UTF8&s=books&qid=1259274738&sr=8-7.

```
            ToCoordinates,
            DeliveryTime) fnot
        delivery_bid(
            DeliveryRequestId,
            _DriverId,
            _CurrentCoordinates,
            _PossiblePickupTime).
no_bid_alert(DeliveryRequestId)<-
     start_manualAssignment(
            DeliveryRequestId,
            StoreId,
            ToCoordinates,
            DeliveryTime) fnot
        delivery_bid(
            DeliveryRequestId,
            _DriverId,
            _CurrentCoordinates,
            _PossiblePickupTime).
```

Manual assignment ❷

This assertion is intended to derive the `No Bidders Alert` event (called `no_bid_alert` in this example). It identifies two cases: the automatic assignment case ❶, and the manual assignment case ❷.

10.1.3 Development environments

There are two types of development environments: text-based and graphically based. These two are not mutually exclusive, as development environments can consist of a mixture of graphical and text-oriented tools. The various environments reflect different assumptions about developers' preferences. In some cases developers prefer a

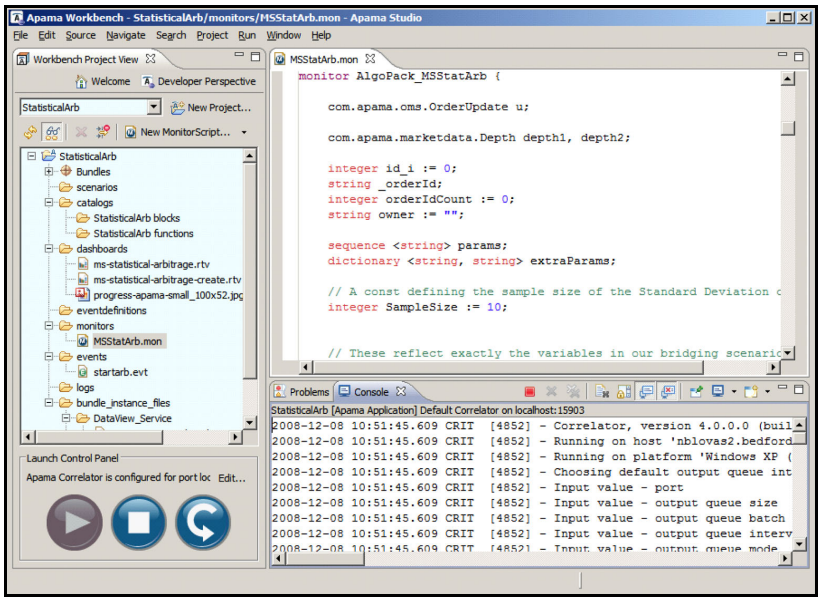


Figure 10.5 A text-based development environment (Apama Studio)

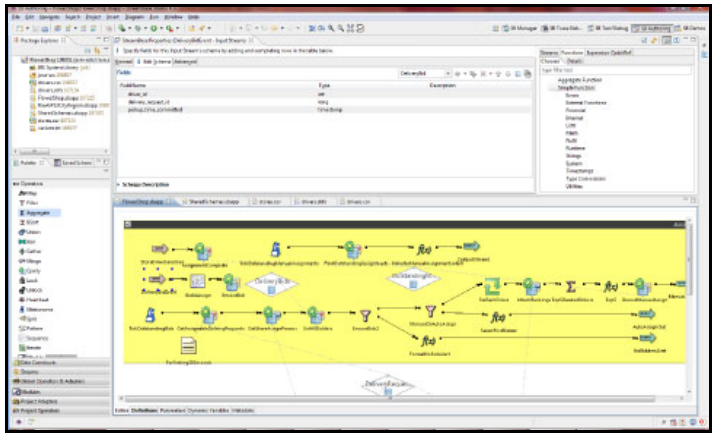


Figure 10.6 Combined graphical- and form-based development environment

more familiar text-based interface, whereas others prefer a more visual style of development. Text-oriented tools can provide a "fill a form" type of interface, as shown in figure 10.4, or they can offer full text entry, like the example in figure 10.5. This is taken from Apama's Eclipse-based IDE, which is called Apama Studio.

StreamBase also has an Eclipse-based IDE, but as you can see from figure 10.6, this has a graphically based development environment, with some functions being provided in a textual manner. In this tool (StreamBase Studio) the EPN is constructed graphically, while event types and individual functions are then built using form-oriented text.

The environments we have looked at here are geared mainly towards technical developers. In chapter 12 we discuss the trend towards having semi-technical event processing developers.

The language and development environment is just one facet of the event processing implementation; next we discuss the non-functional properties of event processing systems.

10.2 Non-functional properties

An important aspect of the engineering and implementation considerations in any system is the non-functional aspect. Non-functional requirements are concerned not with *what* a system does but *how well*. It is often the non-functional properties that make or break a specific application. In this section we briefly survey the main non-functional aspects of event processing systems and explain the particular requirements imposed by event processing systems that the system designer should be aware of. Not all of these requirements apply equally to all applications, so when designing an event processing application one needs to consider which of them are important for the case in hand. In the next section we deal with various optimizations and relate them back to these requirements.

The non-functional requirements that we discuss in this section are scalability, availability, and security. There are further non-functional properties, such as reliability and

usability. We touch on reliability in chapter 11 when discussing inexact event processing, and usability requirements are closely related to the programming styles and development environments discussed earlier in this chapter.

10.2.1 Scalability

We start with our definition of *scalability*.

> **SCALABILITY** *Scalability* is the capability of a system to adapt readily to a greater or lesser intensity of use, volume, or demand while still meeting its business objectives.

Scalability has several dimensions. The dimensions relevant to us here are the volume of events, the number of agents, producers, consumers and contexts, the complexity of computation, and the processor environment.

SCALABILITY IN THE VOLUME OF PROCESSED EVENTS

High event throughput is often considered one of the characteristics and main motivations for the use of event processing software. This is certainly true in some application segments. However, our experience has been that event processing software is employed mainly to increase agility and reduce the total cost of ownership. So the range of applications that are likely to employ generic event processing software is much wider than those that need high event throughput. In these applications scalability means the ability to handle variable event loads efficiently; the quantity of events may go up and down over time.

That being said, some applications require high event throughput, for example, some financial market applications, weather-related event processing, and telephony call tracking. These can encounter extremely high volumes of input events which may require special treatment and optimization. Some systems have been specifically designed with high event throughput in mind, and we discuss performance further in section 10.3.

SCALABILITY IN THE QUANTITY OF AGENTS

In some applications there is a significant amount of processing applied to the events, so the major scalability issue is the ability of the EPN to grow substantially and have a large number of EPAs. An example is a banking system that lets each customer create his or her own sophisticated alerts. Each customer could end up with a unique EPA and this could result in the dynamic creation of a large and complex EPN. When designing an event processing system, estimates about the number of EPAs and their growth curve may impact the way the system is implemented and deployed. Related optimizations are discussed in section 10.4.

SCALABILITY IN THE QUANTITY OF CONSUMERS

In some cases the number of event consumers may become high, for example, the personal banking application that we just mentioned. The event processing system has the challenge of tracking which consumers are active and has to route events to the active consumers and possibly hold them for subsequent delivery to inactive ones. In

some cases a single event emitted from the event processing system may have to be routed to many consumers. This could benefit from optimizations at the routing level, such as the use of multicasting[7] Related optimizations are discussed in section 10.4.

SCALABILITY IN THE QUANTITY OF PRODUCERS

In some cases the number of event producers can grow substantially. Consider a web bookstore that tracks events related to all the customers who browse and buy books to determine patterns of use. If we view every customer as a separate event producer the number of event producers can grow large. Even though the number of events from each customer may be small, the total number of customers can be high and the system has to be able to cope with this. This can also lead to a high number of context partitions, which we discuss next.

SCALABILITY IN THE QUANTITY OF CONTEXT PARTITIONS

In some event processing applications, the number of context partitions that are concurrently active may become large. Consider an internet retail store, with an application which has a context partition that tracks each order from the time it is placed until the items are delivered. Such orders may be numerous, and if we assume that each context partition has an internal state, this requires the event processing system to store a large amount of state information. If each context partition is implemented by a distinct runtime artifact, this also leads to a scale-up in number of these artifacts that the runtime has to manage.

SCALABILITY IN CONTEXT STATE SIZE

Another context-related scalability issue is the ability of a single EPA instance to acquire the space needed to store its internal state, especially if it is associated with a long-running context partition. For example, a sequence pattern running over a 24-hour period might need to accumulate and retain a large number of events each day.

SCALABILITY IN THE COMPLEXITY OF COMPUTATION

The complexity of the EPAs themselves may have substantial impact on the overall performance of the system. Cases where the EPAs implement highly complex logic may require different types of optimization than the other scalability aspects that we have mentioned. We return to this point in section 10.4.

SCALABILITY IN THE PROCESSOR ENVIRONMENT

Event processing systems may run in heterogeneous environments. At one extreme they may run on multiprocessor supercomputers; at the other extreme they may run on small devices that have footprint limitations. Both ends, as well as those in the middle, require specific optimizations, and an implementation that works well at a certain point in this spectrum may need significant redesign to work well on a different size processor.

A system designer should be aware of all these scalability issues when designing an application, as well as the corresponding optimizations discussed in the next section.

---

[7] Multicasting is the ability to transmit a single stream to multiple subscribers at the same time. For more information refer to http://www.tcpipguide.com/free/t_IPMulticasting.htm.

10.2.2 Availability

Availability is one of the notable quality of service requirements in current systems, and we start by defining this term.

> **AVAILABILITY** The *availability* of a system is the percentage of the time its users perceive it to be functioning.

Event processing systems can use existing standard high availability practices like logging, failover, and disaster recovery practices. The designer of an event processing system must, however, make decisions related to high availability. These considerations relate to whether it is cost effective to employ high availability practices, as they have a cost associated with them and they may not be fully required in some applications. An example of such a consideration is the issue of recoverability, as discussed in the sections that follow.

Some event processing agents (such as those that perform aggregation, composition, and pattern detection) are stateful. The internal state of such an agent has to be kept as long as the particular EPA instance is active, meaning as long as its context partition is valid. For example, a sequence Pattern detect EPA running with the reuse policy over a 24-hour window might need to retain all the participant events that occurred during that period. This brings us to the issue of *recoverability*.

> **RECOVERABILITY** *Recoverability* is the ability to restore the state of a system to its exact value before a failure occurred.

If you are interested in learning more about well-known techniques for achieving recoverability, refer to the reading list at the end of the chapter. Recoverability incurs some additional processing overhead as changes in state need to be logged and the entire state needs to be written to a persistent store, at least periodically. This overhead may take a toll on the processing latency and total throughput of processed events. Note that typically states are persisted in checkpoints, and logs are kept between checkpoints.

In some applications recoverability is a must. If the event processing is part of a mission-critical application, and decisions are made using the results of this processing, losing some of the system's state may have critical implications, such as the following: ignoring an order, missing a pattern in a specific customer's behavior, losing the location of a consignment of goods, or making an incorrect decision due to ignorance of a new trend.

For other applications, it might not be cost effective to apply recoverability. Consider a network management system that receives events about observable faults in the system and attempts to find the root cause. Because the events are symptoms of an underlying problem, they will recur anyway until the problem is resolved. In this case recoverability could help identify a problem faster, but it is not vital and might not be cost effective. Likewise, systems which look for statistical trends may be based on sampling or on analysis of a large number of events; in these cases recoverability may not be required.

In conclusion, event processing systems should support recoverability as an optional property with various tuning alternatives (for example, full persistence of state and checkpointing) and the designers of each application should consider the cost effectiveness of recoverability for their applications and decide whether recoverability is required.

From availability we move on to discuss security in event processing systems.

10.2.3 Security

Security requirements relate both to ensuring that operations are only performed by authorized parties, and that privacy considerations are met. Specifically this means the following functions:

- Ensuring only authorized parties are allowed to be event producers or event consumers.
- Ensuring that incoming events are filtered so that authorized producers can't introduce invalid events, or events that they are not entitled to publish.
- Ensuring that consumers only receive information to which they are entitled. In some cases a consumer might be entitled to see some of the attributes of an event but not others.
- Ensuring that unauthorized parties can't add new event processing agents to the system, or make modifications to the EPN itself (in systems where dynamic EPN modification is supported).
- Keeping auditable logs of events received and processed, or other activities performed by the system.
- Ensuring that all databases and data communications links used by the system are secure.

Some people[8] view security and privacy issues as barriers for the trust and utilization of event processing systems. Authentication and authorization issues are a concern because attacks that send false events could be devastating to safety-critical systems, such as those that support air traffic control, or a smart electricity grid. Privacy issues are also a serious concern for people; some people won't install electronic vehicle toll payment devices, such as the E-ZPass[9] system that exists in some of the U.S. states, because they are sensitive to privacy issues and don't want anyone recording information about their whereabouts. Privacy is also a concern in healthcare applications, and in many jurisdictions legislation requires organizations to safeguard the privacy of personal data in all application domains. Trust is particularly significant in applications where sensitive data is passed between different organizations.

---

[8] The view on security and privacy as barriers is taken from Chandy and Schulte's book (K. M. Chandy and W. R. Schulte, *Event Processing: Designing IT Systems for Agile Companies, McGraw-Hill Osborne Media, 1 edition* (September 24, 2009).

[9] http://www.ezpass.com/.

Event processing systems can have various levels of sensitivity to security and privacy issues. If the collection of event producers is a closed set in which security practices are trusted, the problem is reduced. On the other hand, if anybody can be a producer (for example, when events come from Twitter feeds), the security issues may be pervasive.

Studies on related security issues[10] have been conducted, but people mainly deal with security and privacy issues that are specific to event processing in an ad hoc way. The additional reading section at the end of this chapter refers you to material on database security, which shares many common issues with event processing security.

In conclusion, when designing an event processing application, you should be aware that non-functional requirements may have a major impact on the way the system should be implemented and you must make the choices appropriate to your particular application. We now discuss optimization techniques to address some of these non-functional requirements.

10.3 Performance objectives

Some non-functional requirements can be translated to performance objectives which can then be the subject of various optimization approaches. In this section we discuss some of the major performance objectives for event processing relating to throughput, latency, and time-constraint objectives. Table 10.1 summarizes these objectives, and we discuss each objective in this section.

Table 10.1 Performance objectives and their associated metrics

Number	Objective name	Objective metrics
1	Max input throughput	Maximize the quantity of input events processed by a certain system or subsystem within a given time period
2	Max output throughput	Maximize the quantity of derived events produced by a certain system or subsystem within a given time period
3	Min average latency	Minimize the average time it takes to process an event and all its consequences in a certain system or subsystem
4	Min maximal latency	Minimize the maximal time it takes to process an event and all its consequences in a certain system or subsystem
5	Latency leveling	Minimize the variance of processing times for a single event or a collection of events in a certain system or subsystem
6	Real-time constraints	Minimize the deviation in latency, from a given value, for the processing of an event and all its consequences in a certain system or subsystem.

[10] An early article about security issues in pub/sub system is the following: Mudhakar Srivatsa, Ling Liu, *Securing Publish-Subscribe Overlay Services with EventGuard*. ACM Conference on Computer and Communications Security 2005: 289-298 http://portal.acm.org/citation.cfm?doid=1102120.1102158.

All these objectives are intended to address scaling issues, but each addresses them using different assumptions and may be served by different optimizations. As you can see from table 10.1, each objective may apply to an entire system, or to any part of a system. In some systems there is a single performance objective for all the processing in the system, for example, latency leveling for each event type in that system. In other systems there may be mix of performance objectives; some of the events may have real-time constraints associated with them, whereas others may have another metric. Performance objectives may also be composed of several separate metrics. We now briefly discuss each of the six performance objectives defined in table 10.1 and then move on to metric composition.

MAX INPUT THROUGHPUT

This is the performance metric most often mentioned as a motivation for high performance stream processing systems.[11] This metric is strongly related to the requirement for scalability in the quantity of events. This metric measures the number of input events that the system can accept within a given timeframe while continuing to function correctly. It is sometimes referred to as *events per second*. Note that although this metric asserts that the system can absorb events, it does not say anything about the latency of processing. To specify a required latency, this metric has to be composed with a latency-oriented metric.

MAX OUTPUT THROUGHPUT

This is a performance metric that refers to the output throughput rather than the input throughput. It is also measured in events per second, but in this case the measure relates to the number of events that the system generates and not to the input events.

MIN AVERAGE LATENCY

This is the first of the latency metrics. It is a statistical metric that refers to the average latency of all events, and it is measured as a time unit (for example, 10 milliseconds). Because different event types may have different levels of processing complexity, it's sometimes useful to measure the latency of a single event type, rather than the overall metric, which is the average of all the average event type latencies.

MIN MAXIMAL LATENCY

This metric relates to the maximal latency for a certain event type or collection of event types. Note that this is a different objective than the previous one, and there are optimizations that improve one of these metrics, and make the other one worse.

LATENCY LEVELING

This metric is also known as a *deterministic performance* metric, and is sometimes identified with real-time processing. This metric is used by applications that need predictable and low variance performance processing for each event type or collection of event types.

---

[11] An example of an article showing an optimization related to this performance metric is the following: Joel L. Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer: *SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems*. Middleware 2008: 306-325. http://www.springerlink.com/content/9h772844u5875757/.

REAL-TIME CONSTRAINTS

Although latency leveling is identified with real-time systems, these systems may also need to impose particular performance upper limits for either processing of a certain event type, or a certain EPA. This can be achieved through a real-time constraints metric that specifies just such an objective. Note that real-time constraints may be hard real-time, in which case compliance with these constraints is a must, because lack of compliance may have disastrous consequences, or soft real-time constraints that are considered quality of service goals.

COMPOSING METRICS

In some cases there is a need to develop a performance objective that includes more than one metric. This composition may be related to a specific part of the system; for example, there might be an event type which has both throughput- and latency-related metrics. Alternatively, there could be different performance metrics for different parts of the system. An optimization plan might have to take into account different objectives, with some weight factors applied to them, when creating an objective function.

This takes us to the various types of optimization available to help meet such objective functions.

10.4 Optimization types

In this section we discuss various types of optimization that have either been used or have been proposed for use with event processing systems. These can serve as building blocks for an optimization plan that is particular to a specific performance function. We discuss optimizations in the following areas:

- Optimizations related to EPA assignment: partitioning, parallelism, distribution, and load balancing
- Optimizations related to the coding of specific EPAs: code optimization and state management
- Optimization related to the execution process: scheduling and routing optimizations

It should be noted that the optimization considerations are quite complex, and this area is still in need of more established methods and practices. The purpose of this section is to make applications designers aware of optimization opportunities, rather than to provide a recipe to optimize a specific application.

10.4.1 EPA assignment optimizations

In chapter 6 we stated that an event processing agent represents a logical function, and that there are various ways to map these logical functions to physical runtime artifacts. This is the basis for EPA assignment optimizations, as the choice of assignment can influence the performance metrics that we listed earlier.

These optimizations are also known as *black box* optimizations, because the EPA's implementation is assumed to be fixed. They deal with factors external to the EPA

such as its location and relative scheduling. The following sections survey the most common assignment optimizations.

PARTITIONING OF EPA INSTANCES TO RUNTIME ARTIFACTS

The way that EPA instances are mapped to runtime artifacts can have a major effect on the various performance metrics. We refer to this as *partitioning* the EPAs, and the idea is to group EPA instances so that they execute together for better performance. The two extremes are one where there's a single centralized runtime artifact that embeds all the EPA instances, and one where there's a separate runtime artifact for each EPA instance. The centralized solution has benefits for cases where the volume of events is not an important measure, because it saves the overhead of communication between the different EPAs. Partitioning is the key both to parallel execution and to distributed execution.

Partitioning decisions can be driven by the EPN topology as this determines the dependencies between the EPAs, although some languages, such as SPADE, let the programmer make partitioning decisions. One approach to partitioning is based on assigning EPAs to strata, where the EPAs in each stratum are independent of one another and can run in parallel. If EPA1 produces events that are consumed by EPA2, then EPA2 is placed in a higher stratum. We show an example of stratification in figure 10.7, where the EPN is partitioned into three strata, each of which contains independent EPAs.

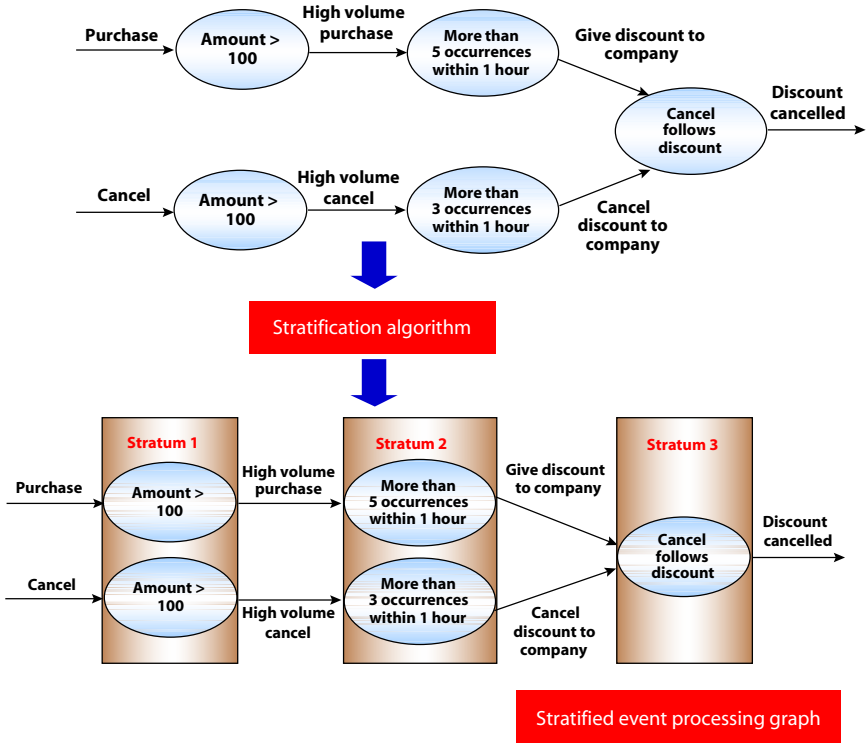


Figure 10.7 Stratification of an EPN to three strata

Note that this is a simple example, and for EPNs in which there are many interdependencies between EPAs the stratification process is more complex. We reference an article describing stratification-based optimization at the end of this chapter.

This stratification process tells us which EPAs can run in parallel, but to decide which of them should be grouped together in the same runtime artifact we have to consider other factors, such as the number of available cores/processors, the level of distribution, the communication overhead, and of course the performance objective function. We discuss some of these aspects in the sections that follow.

PARALLEL PROCESSING

One of the major ways to achieve various performance metrics is parallel processing. There are three levels of parallelism: first, parallelism inside a single core using multithreading; second, parallelism by partitioning the work within a multicore machine where the threads running in different cores have access to shared memory; and third, partitioning the work to multiple machines within a cluster. Decisions on which activities should be run in parallel are difficult, and are usually made automatically by a system optimizer, rather than being performed manually. You may find it interesting to look into research performed on such parallel processing.[12]

DISTRIBUTED PROCESSING

An additional optimization method involves moving the processing close to the producers and consumers where applicable. Consider an example where there are multiple sensors within the same location, and the event processing involves aggregation of events that are emitted by these sensors. Placing the aggregation EPA close to the sensors can eliminate a substantial amount of network traffic. Likewise, if the EPN contains an EPA that creates many events that are all consumed by a certain consumer, or a set of consumers that are located in a certain location, it might be useful to locate this EPA close to the consumer or consumers. This optimization approach can also complement the parallel processing approach. If the parallel event processing is executed over a grid of machines within various geographic locations (instead of being on a physical cluster or co-located set of multicore machines) it might be sensible to co-locate a group of agents if there's a substantial amount of communication between them.

LOAD BALANCING

Static optimization techniques, such as stratification, involve analysis of the EPN dependencies, making some assumptions about the traffic load and available resources. However, these assumptions, as well as the topology of the EPN, may change over time. The introduction of more resources, the temporary unavailability of computing resources, as well as unexpected changes in the distribution and load of events, are all reasons for reevaluating the partitioning scheme. Answering the general question of how and

[12] An example of such optimization for stream processing is in the following article: Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel L. Wolf, Kun-Lung Wu, Henrique Andrade, and Bugra Gedik, *COLA: Optimizing Stream Processing Applications via Graph Partitioning*,Middleware 2009: 308-327. http://www.springerlink.com/content/aw817m13m4536001/.

when to rebalance the load in this way requires more work, although there are some ad hoc solutions in use today.

Another approach, used in high throughput event applications, is to discard events if there aren't sufficient resources available to process them all. This form of load balancing, which results in approximate event processing, is called load shedding.[13] The book by Chakravarthy and Jiang that is referenced at the end of this chapter surveys load shedding techniques used in stream processing.

Some performance objectives require us to go further than the *black box* approaches we have discussed so far, and optimize the actual EPA code itself. We discuss a couple of such *white box* optimizations next.

10.4.2 EPA code optimizations

White box optimizations are optimizations that modify the internal execution of EPAs. This area is less developed than the black box optimizations. We briefly discuss some of the possibilities in this area starting with code generation and then moving on to the more developed area of state management.

OPTIMIZED CODE GENERATION

Query optimization is a vital part of relational database execution; substantial research and development have been invested over the years in this area. The core idea behind query optimization is that although queries may look similar, different queries have different optimized execution plans, and thus an optimizer might generate totally different code.

The equivalent of this idea is also valid for event processing, and you might hope that if you have a language that is an extension of SQL, you could adjust the SQL query optimization to include continuous queries. However, it turns out that these adjustments are not trivial.

To see why this might be not trivial, consider the sequence Pattern detect EPA shown in figure 10.8.

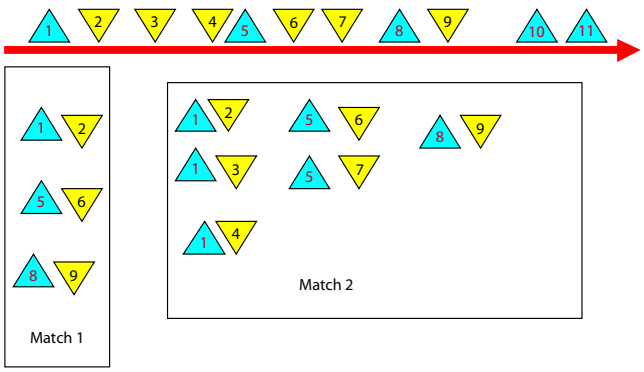


Figure 10.8 A sequence Pattern detect example showing the results of two different policies

[13] For example, refer to the following article: Nesime Tatbul, Ugur Çetintemel, and Stanley B. Zdonik, *Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing*, VLDB 2007:159-170. http://www.vldb.org/conf/2007/papers/research/p159-tatbul.pdf.

In this example, there are two types of events. The first type (E1) is shown as a triangle pointing upwards, and has five instances: 1, 5, 8, 10, 11. The second type (E2) has six instances: 2, 3, 4, 6, 7, 9. These instances arrive in the order shown at the top of the diagram, and the pattern is looking for the sequence <E1, E2>. As we saw in chapter 9, the output of a Pattern detect EPA depends on the matching policies being used. Suppose that the EPA uses the immediate, override, and consume policies. This means that it matches each `E1` event with the next occurring `E2` event to produce three matching sets: <1, 2>; <5, 6>; <8, 9>. However, if it were to use the reuse policy rather than consume it would match each `E1` event with multiple subsequent `E2` instances to create the six matching sets shown as "Match 2" in figure 10.8. These two results are quite different, and it is conceivable that the optimal data structure and implementation code will also be different. Code optimization should take account of this.

Another optimization that is being used by some event processing implementations is the use of Real-time Java,[14] which allows for thread priorities and smoothes the memory management.

STATE MANAGEMENT

State management optimization relates to the way that an internal state is held by EPAs, and in some cases also to global state elements. The basic trade-off is between performance and recoverability. Memory-based state provides better performance, but recoverability requires some overhead and implementation complexity. Persistence-based state (for example, one where state is held in a database) provides better recoverability, but may conflict with performance goals. The implementation of state management is a function of the requirements, both for performance and recoverability. Use of in-memory state can also be problematic when there is a need for scalability in the number of context partitions or in the quantity of events accumulated within a context partition.

The choices in this area include the following:

- Using disk-based persistence. This resolves space scalability issues and recoverability; however, it may harm performance goals.
- Using in-memory databases that provide caching capabilities while guaranteeing recoverability. This is a way to balance between the two sides of the trade-off. There are various tuning possibilities that can be made, based on assumptions about mean time between failures (MTBF) and mean time to recovery (MTTR).
- Using grid memory instead of persistence. The idea here is to replicate the state in memory held on multiple machines, to get recoverability without having to use disk-based persistence. This solution has an overhead of network traffic, and the complexity of synchronizing among the different replicas.
- Using a mixture of these approaches, for example, persistent storage for states that have space scalability issues, and in-memory for others. You can also allow different levels of recoverability for different EPAs.

[14] http://www.rtsj.org/

We complete our survey of optimization techniques with a discussion of execution optimization.

10.4.3 Execution optimizations

Some additional optimizations can be performed at execution time.

SCHEDULING

Scheduling optimization deals with planning the order of EPA execution, in cases where there is no natural order of precedence, but the EPAs concerned compete with each other for computing resources. Scheduling optimizations can be done when there are different performance requirements for different EPAs, for example, when one EPA has real-time constraints while the other does not. In such cases you might use a preemptive schedule that delays the execution of a runtime artifact that has already started in order to execute another runtime artifact that needs to run to comply with its real-time constraints. Scheduling can also be done by analysis of the EPN topology, giving priority to EPAs that are in a critical path to achieve performance criteria.[15]

ROUTING

Various optimizations that relate to the transport layer deal with the manner and physical implementation of routing between the various components of the systems (producers, EPAs, and consumers). This relates to the way that event channels are implemented and the routing method used (anycast, broadcast, multicast, and unicast). Routing optimizations are typically assumed to be the role of the transport infrastructure.

There is no comprehensive methodology for event processing optimization, so performance tuning of event processing applications is still an art rather than a precise science. This section provided a quick look at some techniques that a system designer can use in order to optimize an application. Some of these optimizations are provided, to some extent, by today's event processing platforms. However, this is still an active area of research and development, and we expect that more optimization tools and methodologies will be provided in the future. Next, we move from optimization to our last engineering topic: event processing validation.

10.5 Event processing validation and auditing

We have noted that event processing programming differs in some ways from regular programming and thus needs its own validation tools. Validation consists of static analysis and dynamic analysis of event processing networks. In essence, these analyses provide observations about EPNs that point out possible problems in the design of event processing applications. We briefly describe some of the main observations that can be obtained from such analyses and conclude this section with a discussion of auditing of event processing applications.

---

[15] The following article shows an optimization related to scheduling: Joel L. Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer, *SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems*, Middleware 2008: 306-325. http://www.springerlink.com/content/9h772844u5875757/.

10.5.1 Static analysis of event processing networks

Static analysis observation is analysis that is performed on the EPN model, as described by our building block language. It can serve to validate the design of an event processing application. In some cases observations at the static analysis level can point out possible problems that might not actually happen in reality; thus, to complete the validation picture you need to complement it with dynamic analysis (this is analysis that examines what happens at runtime). It should be noted that static analysis can be used in the design phase, and also as part of change management, where it can be used to check whether a proposed change might introduce a problem.

TERMINATION PROBLEM OBSERVATION

A termination problem is a case in which the system (in this case, the flow of events in the event processing network) does not terminate due to an infinite loop. Recall that the edges in the EPN denote events that are emitted by one node in the network and serve as input to another node. The event processing network may be cyclic. For example, an event that is derived by EPA1 serves as input to EPA2, which derives an event that serves as input to EPA1. This is a simple cycle that consists of two EPAs, but a cycle can be a longer cycle that consists of multiple EPAs. A cycle in the EPN may or may not indicate a design problem. It may indicate a design problem if indeed it creates an EPN that never terminates where this was not the intention of the designer. It may not be a problem, if the system is really cyclic in nature, like a state machine that always returns to the initial state after getting to its final state. It may also not be a problem if this infinite loop is only theoretical, and the combination of conditions in which it occurs can't occur in reality or doesn't occur even if it could.

A validation tool can find cycles in the EPN. In some cases it can also check whether the conditions in the cycle have internal contradictions, so that this cycle can't occur in reality, and make this observation visible to the designer. Note that the general issue of discovering whether a set of conditions can be satisfied together is an NP-complete problem, but in some cases this can be detected more easily.

Figure 10.9 illustrates an EPN containing five EPAs, called R1,…,R5. There are two cycles, a smaller cycle consisting of R2 and R3, and a bigger cycle consisting of R2, R3, R4, and R5. Figures 10.9, 10.10, and 10.11 are taken from an event processing validation research project done in the IBM Haifa Research Lab by Ella Rabinovich and Sarit Arcushin.

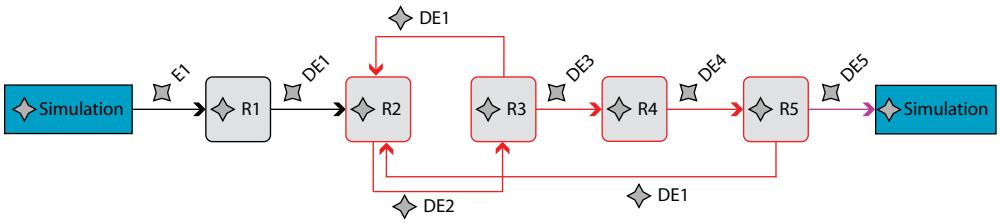


Figure 10.9 An example of two loops detected within a path in EPN

EVENT NOT USED

This observation discovers events that are produced and not used. A producer may produce an event that is not consumed by any consumer or EPA. An EPA may derive an event that again is not consumed by any EPA or consumer. This may indicate a design problem, or it could be caused by a change that results in a certain event no longer being required. Events that are not used can be observed by a static analysis tool and be flagged up to the user of the tool.

EPA IS NOT REACHABLE

This type of observation traces the fact that an EPA never executes because one or more of its input events never flows. This may be a design flaw or the result of a change that eliminates some event from being produced, derived, or routed to the EPA. This observation can be obtained by static analysis of the EPN, and the EPN designer can be alerted.

EPA IS A DEAD END

This type of observation identifies the fact that an EPA does not produce any derived events. This may be a design flaw or the result of a change. Again this observation can be obtained by static analysis of the EPN, so as to warn the designer. Figure 10.10 shows an example of a dead end; this is an EPN with several consumers, producers, and EPAs. The marked EPA, whose label is `"Handle Low Inventory with No PO"`, does not have outgoing edges, which means it does not produce any derived events.

POSSIBLE NONDETERMINISTIC PROCESSING

This type of observation identifies the fact that two or more EPAs can execute in any order, and may yield different results depending on their order of execution. This can

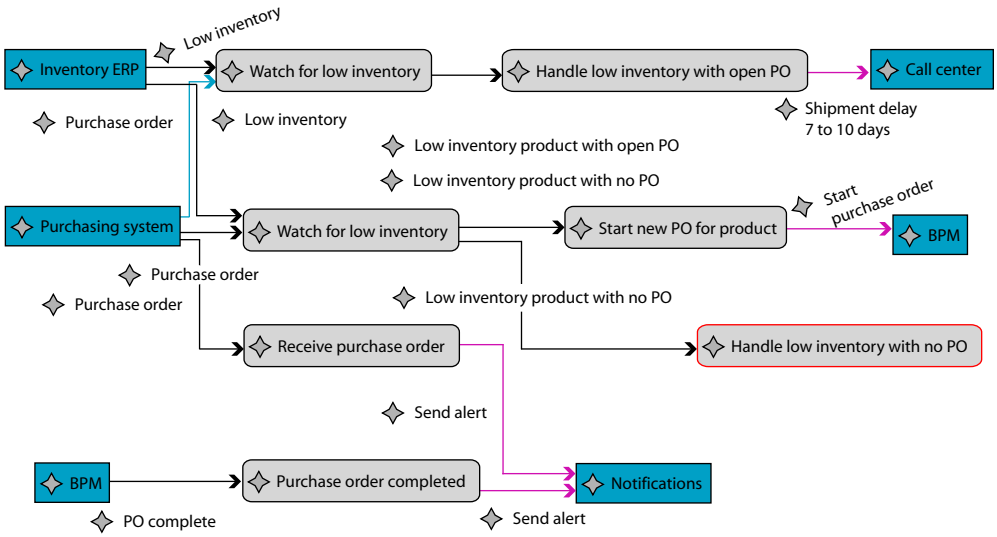


Figure 10.10 A dead end example. The EPA `Handle Low Inventory with No PO` does not have any outgoing events.

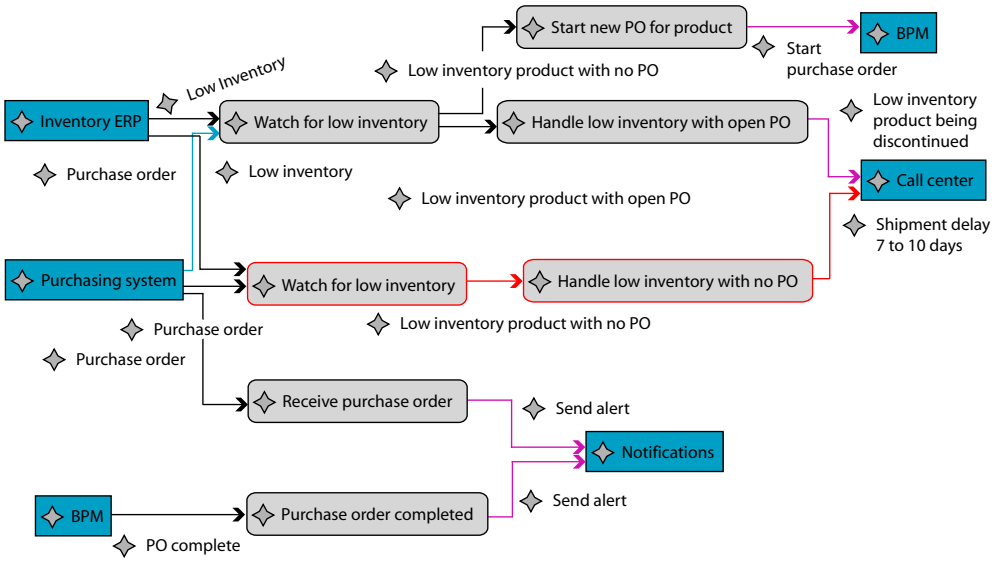


Figure 10.11 An example of provenance—finding the path in the EPN that contains all the antecedents of the derived event `"Shipment Delay 7 to 10 Days"`

serve as an indication to the designer to specify priority ordering of those EPAs, if this is supported by the event processing system, or to redesign the EPN flow, should this nondeterministic order actually matter to the application.

Validation tools may also provide information about event provenance. They can operate forwards, showing all consequences of a certain event or node in the EPN, or backwards, showing all antecedents of a certain derived event or node in the EPN. Figure 10.11 shows an example of tracing back to see the path in the EPN that leads to the fact that a derived event `"Shipment Delays 7 to 10 Days"` has been obtained.

Static analysis tools may provide textual or graphical output. They are complemented by dynamic analysis techniques, which are discussed next.

10.5.2 Dynamic analysis of event processing networks

Dynamic analysis is an analysis that is based on observation of runtime execution and not on the static model. There are two types of dynamic analysis for EPNs: simulation and tracing. Simulation creates simulated events and scenarios and runs the application using this simulation, which may provide observations. Tracing takes traces of runtime executions and analyzes them. Dynamic analysis may provide observations that can't be obtained in static analysis. We briefly explain some of these observations.

ACTUAL TERMINATION ISSUES

Whereas static analysis indicates the possibility of termination issues, dynamic analysis may detect that a termination problem actually exists. In the simulation mode we can observe that a loop is actually realized; in tracing mode we can observe that loops indeed happened.

ACTUAL REACHABILITY ISSUES

An EPA may be reachable, as far as the static analysis is concerned, and there is a visible path in the EPN to the EPA; however, this EPA is never activated in the simulation or in a trace.

OUTPUT TERMINAL IS UNUSABLE

The events that flow out of an output terminal of a certain producer or EPA could be unusable, in the sense that none of them is consumed by an EPA or consumer. This may be an indication that the events produced by this EPA or producer can be eliminated.

In addition, dynamic analysis can provide actual provenance for a specific instance of an event or any EPN node, backwards or forwards, and observation about quantities of raw and derived events that may be used in optimization decisions. Validation and debugging tools are important to making new paradigms usable in practice.

Our final engineering topic concerns the auditing of an event processing application.

10.5.3 Event processing auditing

Auditing is the ability to investigate whether processes have been applied in an appropriate way. This may refer to whether a process complies with external regulations, or internal policies, or whether a decision has been made in an appropriate way. Whereas validation is intended to validate the implementation, auditing is intended to validate the usage of the system and the work processes behind a certain information system. In chapter 5 we stated that one of the event consumer kinds is an event log which retains raw or derived events for further processing. The events persisted in the event log may be used as the audit trail that is required for performing auditing functions.

The audit itself is done by querying this event log. Queries may be similar to the dynamic analysis we have just discussed. The kind of query that can be used depends on the amount of information that is stored in the event log, alongside the events themselves. If the entire path in an EPN is retained, queries like the following can be made:

- Trace all antecedents of a certain event or EPA instance activation
- Trace all consequences of a certain event.

Other types of audit queries may require temporal queries of the type demonstrated by the following query:

- Since the beginning of 2010, have all the manual assignments made by the Exotic Flowers store used only the same five drivers?

This type of query may be more easily answered if the event store is a temporal database. You can find more material about temporal databases in the additional reading section at the end of this chapter.

This ends our engineering-oriented discussion and is a good time to summarize this chapter.

10.6 Summary

In this chapter we've discussed some of the engineering aspects of event processing. We looked at software engineering and reviewed various programming styles and development environments. We then talked about non-functional aspects of event processing, followed by a discussion of performance objectives and optimization techniques, and event processing validation. Current engineering practices provide solid foundations for many existing applications, but as the area of event processing is evolving, its software engineering aspects and the various optimization techniques will also evolve. The next chapter discusses challenges within the current state of the practice.

10.6.1 Additional reading

Schmidt, Klaus. 2006. *High Availability and Disaster Recovery: Concepts, Design, Implementation.* Springer. http://www.amazon.com/High-Availability-Disaster-Recovery-Implementation/dp/3540244603/ref=sr_1_2?ie=UTF8&s=books&qid=1259392209&sr=8-2. This book is recommended if you want to fully understand high availability techniques.

Bernstein, Philip A., Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison Wesley. The book can be freely downloaded from Phil Bernstein's homepage: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx. This is a classic book and an excellent source for anyone who wants to understand the principles of recoverability and the techniques needed to implement it.

Ben-Natan, Ron. 2005. *Implementing Database Security and Auditing: Includes Examples for Oracle, SQL Server, DB2 UDB, Sybase.* Digital Press. http://www.amazon.com/Implementing-Database-Security-Auditing-Examples/dp/1555583342/ref=sr_1_2?ie=UTF8&s=books&qid=1259399581&sr=1-2. This book deals with the state of the practice in database security, and provides insights on approaches to security issues.

Chandy, K. M., and W. R. Schulte. 2009. *Event Processing: Designing IT Systems for Agile Companies.* McGraw-Hill Osborne Media, 1 edition. http://www.amazon.com/Event-Processing-Designing-Systems-Companies/dp/0071633502/ref=sr_1_1?ie=UTF8&s=books&qid=1258816511&sr=8-1. This book, which we have mentioned before, is included here as it describes security and privacy issues as barriers to the adoption of event processing (chapter 12: The Future of Event Processing).

Liu, Jane W. S. 2000. *Real-Time Systems.* Prentice Hall. http://www.amazon.com/Real-Time-Systems-Jane-W-Liu/dp/0130996513/ref=sr_1_5?ie=UTF8&s=books&qid=1259439073&sr=1-5. This book explains the basic concepts of real-time systems.

Lakshmanan, Geetika T., Yuri G. Rabinovich, and Opher Etzion. "A stratified approach for supporting high throughput event processing applications." DEBS 2009. http://portal.acm.org/citation.cfm?doid=1619258.1619265. This article describes partitioning of EPAs using the stratification approach.

Chakravarthy, Sharma, and Qingchun Jiang. 2009. *Stream Data Processing: A Quality of Service Perspective: Modeling, Scheduling, Load Shedding, and Complex Event Processing.* Springer. http://www.amazon.com/Stream-Data-Processing-Perspective-Scheduling/dp/0387710027/ref=sr_1_1?ie=UTF8&s=books&qid=1259477906&sr=1-1. This book pro-

vides an introduction to stream processing, and discusses several load shedding and scheduling optimizations.

Etzion, Opher. "Reasoning About the Behavior of Active Database Applications." Rules in Database Systems 1995: 86-100. http://www.springerlink.com/content/f81uw237j1151663/. This paper provides discussion on validation in active database applications and explains some of the concepts being discussed in the event processing validation section.

Etzion, Opher, Sushil Jajodia, and Suryanarayana Sripada (editors). 1998. *Temporal Databases: Research and Practice*. Springer. http://www.amazon.com/Temporal-Databases-Research-Practice-Computer/dp/3540645195/ref=sr_1_1?ie=UTF8&s=books&qid=1262847152&sr=8-1. This book provides a collection of articles about various aspects of temporal databases, including a glossary.

10.6.2 Exercises

10.1 How do continuous queries and rules relate to the concept of an EPA?

10.2 In the EPN model we present in this book, we associate processing functions with the EPN nodes. Some stream processing representations associate functions with edges. Can you describe an alternative EPN representation in which the functions are associated with edges? Can the functionality be spread between nodes and edges?

10.3 What are the pros and cons of graphical- versus text-oriented development environments?

10.4 State the non-functional requirements, performance metrics, and optimizations for the Fast Flowers Delivery application used in this book.

10.5 Devise guidelines for using the various performance metrics that we list.

10.6 Which of the optimizations mentioned can be controlled by an application designer, and which depend on capabilities provided by event processing middleware?

10.7 Are the various event-processing programming styles and non-functional requirements related or totally orthogonal to each other? Provide examples to justify your answer.

SOFTWARE ENGINEERING

Event Processing IN ACTION

Opher Etzion • Peter Niblett

Event processing apps collect, analyze, and react to events as they occur. They recognize event patterns—from the obvious to the complex, even predicting outcomes such as power shortages or customer dissatisfaction—and respond to them accordingly. In some applications, such as financial trading, fast reaction times are a must.

Event Processing in Action is a ground-breaking book that shows you how to use, design, and build event processing applications. It follows a detailed example to present the concepts and show you the how-tos of both architecture and implementation. The book and its accompanying website introduce the leading free and commercial tools available, along with several language implementations and many examples.

What's Inside

- Event processing concepts and applications
- The event-driven application lifecycle
- How to fit event-driven architectures into your enterprise apps
- Things to consider in your implementation

This book is written for software architects and developers. It requires no previous knowledge of event processing.

Dr. Opher Etzion is the chair of the Event Processing Technical Society and leads the Event Processing team at IBM's Haifa research lab. An IBM senior architect, **Peter Niblett** led IBM's work on the JMS interface definition, and chaired the OASIS Web Services Notification committee.

For online access to the authors and a free ebook for owners of this book, go to manning.com/EventProcessinginAction

"All you need to start building useful software."
—From the foreword by David Luckham

"A progressive approach, with pragmatic examples."
—Christophe Avare, Thales Research & Technologies

"Makes this complex subject very manageable."
—Tony Niemann
Zier Niemann Consulting

"Event processing distilled."
—Paul Benedict
Argus Health Systems

"A richly illustrated deep dive."
—David Dossot
Pug Pharm Productions Inc.

ISBN 13: 978-1-935182-21-4
ISBN 10: 1-935182-21-8
54999
9 781935 182214

MANNING $49.99 / Can $62.99 [INCLUDING eBOOK]