TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

M A G I S T E R A R B E I T

Efficient Indexing, Search and Analysis of Event Streams

EventCloud: Exploration and Analysis Tool for Events

Ausgeführt am Institut für

Software Technology and Interactive Systems

der Technischen Universität Wien

unter Anleitung von

Dipl.-Ing. Dr. Alexander Schatten

Ao. Univ.-Prof. Mag. Dipl.-Ing. Dr. Stefan Biffl

durch

Roland VECERA, Bakk.techn.

Sindelfingenstraße 6

A-2000 Stockerau

Ort, Datum

Unterschrift

Start with a basic question, *is life simple*? Most people, if they think about that question, will truthfully answer, *"no"*... Events happen in life that are neither simple in *how* they happen, nor in the *effects* they have.

David Luckham

For my parents.

Abstract

Today's business is event-driven. Businesses of all shapes and sizes are driven by and respond to events. Thus IT-Systems offer a large amount of events, representing notable activities in complex workflows and business processes, that are the subject of event analysis. This thesis highlights the necessity for specialized event analysis tools, explains the relationship to Complex Event Processing (CEP), outlines requirements and proposes the EventCloud system.

EventCloud is a generic, offline event analysis tool for Complex Event Processing (CEP) scenarios. EventCloud enables its user to navigate through events, pick up single events and display their content, discover chains of events and how they are correlated, and to recognize patterns inside the events. Metrics can be defined to provide insight at a higher level and to measure business performance. It visualizes business and process developments at the fine-grained level of events.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

February 8, 2007

Contents

1 Introduction

1.1 Importance of Events in Today's Business

Today's business is event-driven. Businesses of all shapes and sizes are driven by and respond to events. To reflect this behavior, we currently see a shift from a request/response or a pull-based paradigm to a push-based paradigm as proposed by event-driven architecture (EDA).

NGEs(Next Generation Enterprises) rely on automation, mobility, real-time business activity monitoring, agility, and self-service over widely distributed operations to conduct business [34]. Since Gartner proclaimed the term "zero-latency enterprises" [30], enterprises strive to become "real-time" where all notable events of a business are captured, analyzed and responded to immediately. To enable such enterprises, current technologies like Data Warehousing (DWH) and Business Intelligence(BI) are not sufficient as they do not provide adequate support for real-time and closed loop decision-making[15].

Gartner states that events have been underutilized until lately, but are now widely used in business application as they allow to optimizing business processes by cutting costs and improving responsiveness [31]. This follows to a new generation of emerging software that act on low-level (business) events and generates higher-level, complex events to sense opportunities, business situations and exceptions, and perform, if needed, accurate adjustment actions.

The term Complex Event Processing (CEP) was coined by Luckham in [22] and defines a set of technologies to process large amounts of events and utilize these

events to monitor, steer and optimize the business in real-time. CEP is a core enabler for business activity monitoring (BAM) solutions that monitor the relevant business events for changes and indicate opportunities or problems to business analysts.

While CEP solutions offers considerable advantages for enterprises and allows them to come closer to the goal of zero-latency, CEP also brings along new complexity in the IT-landscape of a company. In this thesis I will concentrate on the immense importance of the data processed by a CEP system: the events. Business events, that are the input for CEP, as well as action events that are triggered by the CEP must be stored and collected for later analysis and retrieval to answer questions and gain insight into the enterprise's business operations.

1.2 Complex Event Processing

Complex Event Processing (CEP) introduces an enhanced analysis view on business processes and allows new approaches for business intelligence solutions. Event streams processed in real-time can be used to sense and respond immediately to current business situations and to take advantage of time-sensitive business opportunities.

CEP is applied in many areas with large amounts of real-time data where the delay to take an action must be reduced to a minimum. CEP allows to minimize all three types of latency introduced by [13] - data latency, analysis latency and decision latency - as all three steps can be executed by CEP solutions automatically. Typical application areas are stock trading applications, fraud detection and monitoring complex business processes.

CEP applications generate additional knowledge from a stream of fine-grained events by analyzing related events and calculating data for a higher level view, sometimes called a "complex event". A system implementing CEP must process large amounts of events from multiple streams in real-time and therefore requires

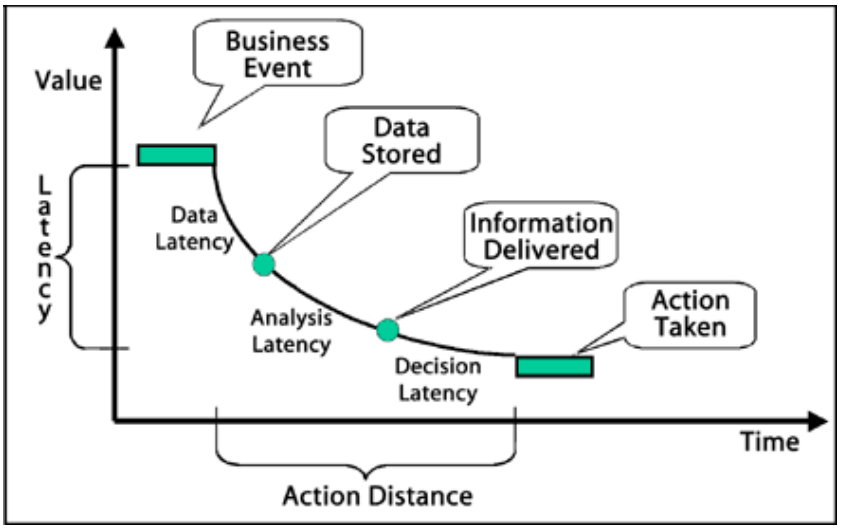

Figure 1.1: Use CEP to minimize latency to react on business event [13]

a high degree of performance. High availability as well as QoS is needed to provide a 24*7 infrastructure for real-time event processing. Besides this technical aspect, and at least equally important for the successful integration of CEP software, highly specialized domain expertise is necessary to properly implement complex use cases within a CEP software.

Many different software architectures were proposed to meet the technical requirements of event stream processing. Continuous queries [2] and event stream processing (ESP) [6] [1] coming from the area of active databases were introduced, event pattern languages [23] and rule engines based on events were created, as well as the Sense and Response paradigm for event processing [15]. Schiefer and McGregor propose in [29] an architecture which correlates events of a business process and applies user-defined functionality on them.

While Complex Event Processing needs a technically mature software architecture and was the scope of many research projects, domain experts are still needed to adopt a CEP solution in a specific business area to really utilize the power of CEP. Therefore CEP projects are not trivial to implement: domain experts first need

to transform domain knowledge into complex queries and rules in the language of the CEP software before they can be applied continuously on the stream of events during runtime. It becomes clear that CEP is only significant if its domain-specific configuration allows a business to perform better than without CEP.

Once a CEP is configured for an use case, it is very likely to degenerate to a black box. If it was properly set up the system can provide added value for the case, however this can dramatically shift over time since business processes are clearly subject to change. Input events change and business rules for the event stream may no longer be accurate. Maybe there are better rules than the ones currently applied to the event stream or even worse: important event patterns that have a revenue for an use case have been completely overlooked.

1.3 Requirements for an Event Analysis Tools

Tools are needed not only to monitor existing CEP applications but also to analyze events *offline* to get better insight into a business in order to optimize the CEP configuration since adaptive business solutions are only manageable if the business operations stay transparent. [22] describes these *Analysis Tools* as a core functionality of the CEP infrastructure. They consume information, such as events plus additional data, from the CEP solution creating a presentation layer for user interaction. Following Luckham in [22], an analysis tool for CEP must include capabilities for:

Display parameters and attributes of an event A user needs to look at the details of a single event without being overloaded with information of all events.

Represent correlated events By selecting an event, a user asks to show all events that caused this particular event as well as all events that are caused by the selected event. Starting from a single event, this facility supports the exploration and navigation of the event data basis.

Graphical present events, event timelines and event correlation The tool should provide graphical representations to display single events, their timing rela-

tionships and their correlation to each other to help a user to understand how activities happened. Luckham argues that visualization often suggests hypotheses to users, so they are essential for event analysis.

Search Patterns Searching the large basis of events allows filtering for specific instances of events and their correlations. The user needs the facility to retrieve an event result set that matches a given query or pattern. Searching is needed to retrieve (exceptional) situations that are the scope of further analysis and investigation.

Drill-down Drill-down from a higher level view to the fine-grained events that represent the business process must be possible. Higher levels are more easily understandable, but the drill-down to the low-level allows detailed analysis of the events aggregated at the higher level.

EventCloud is a generic *Event Analysis Tool* that can easily be applied on CEP scenarios. EventCloud enables the user to navigate through events, pick up single events and display their content, discover chains of events and how they are correlated, and to recognize patterns in the cloud of events. Metrics can be defined to provide insight at a higher level and to measure business performance. It visualizes business and process developments at the fine-grained level of events.

1.4 Interaction of Event Analysis and Complex Event Processing

In Figure 1.2 we see how EventCloud interacts with a CEP application. Different CEP systems internally handle event processing in different ways. However, they all have an *input stream of events* from multiple source systems which represent relevant, real-world activities in a domain. If a CEP solution uses a complex, layered architecture it creates *internal events* used during event processing. An input event can trigger multiple internal events that are used for decision making inside the CEP software. These events are usually completely hidden from outside the CEP solution. Nevertheless they can be very useful for event analysis, as internal events can help to trace CEP's output for a given event input stream,

because they describe the calculation and decision-making process inside the CEP software. Finally the CEP software generates some *output*. This output can be interpreted as an event, too. The output represents any action such as handling an exception, generating an email-alert or publishing a new event that is again input for the CEP software.

Let's consider examples for the three types of events that might be available in a stock trading CEP application or an application monitoring a TMS[1].

Input Events Real-world events and activities of interest like: *StockTick*-, *TransportStart*-, *OrderReceived*-event etc.

Internal Events Internal Events used inside the CEP solution during the process of decision finding: *SellRecommended*-, *TransportDelayDetected*-event etc.

Output Events Output event representing the observations and decisions the CEP solution made: *SellStock*-, *AlertTransportDelay*-, *ModifyTransportRoute*-event, etc.

All this data is available for CEP but only used for event processing and discarded immediately afterwards. Since events represent complex situations and relevant activities they should be preserved for analysis task in some form. The goal of the EventCloud system is to collect these high-quality by-products of CEP (*input events, internal events, output events*) in the EventCloud repository in such a way to provide the base for event analyzing. To allow search and analysis of events, EventCloud must offer complex services like event correlation (Chapter 3), an event search service with different search levels (Rank1, Rank2 and Rank3 as described in Section 4.2), metrics on correlated events (Chapter 7) as well as an intuitive user interface (Section 5.4).

Analysis tools like EventCloud are essential in any phase of a CEP project. During requirement analysis it can be used to analyze the business events currently available in the domain, how they occur, and how they are related. During development of the CEP use case, EventCloud can be applied to validate the results

[1]Transport Management System

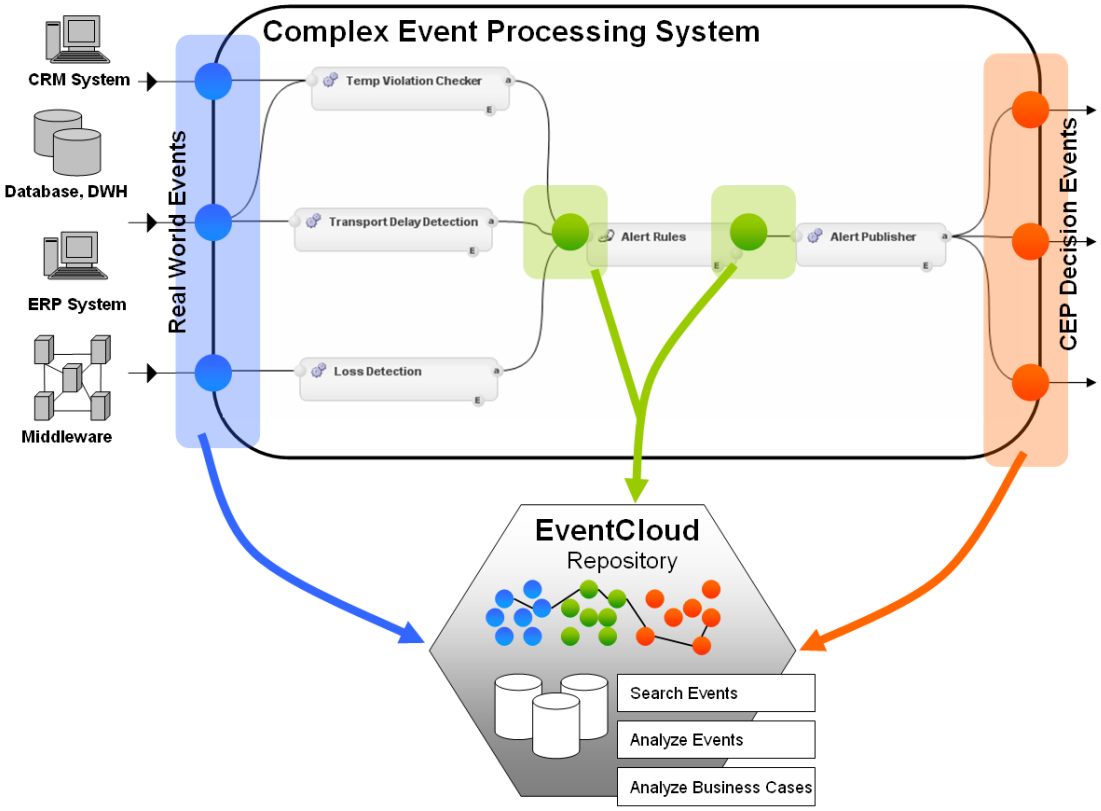

Figure 1.2: CEP software and EventCloud

and decisions made by the CEP. Scenarios can be simulated to see if the domain knowledge is properly mapped into the CEP solution. When the application is deployed, EventCloud can be used to observe the results delivered, to enhance and adopt rules, investigate special situations and to gain domain knowledge.

CEP is not necessarily a requirement to use EventCloud. EventCloud may also be directly applied to a stream of events. In this case, EventCloud correlates events collected from different sources to allow event analysis. Furthermore, not all events used with or generated by CEP needs to be added to the EventCloud as some events may be too fine-grained or redundant and are not useful for event analysis.

2 The Need for Event Analysis: A Motivation

2.1 Medicare Use Case

Rozsnyai presented the Medicare example, a TMS use case for CEP applications, in [28]. SENACTIVE InTime[9] was used to implement a fully CEP scenario which evolved from Rozsnyai's use case description. Figure 2.1 outlines the two event processing maps implemented in SENACTIVE InTime.

In a nutshell, the use case continuously monitors and steers the stock level of multiple stores of a company. If a demand arises at a Location A and can't be fulfilled locally, the CEP application checks the available stocks in other locations if they can satisfy the demand. If so, the best transport (e.g. the cheapest transport) is selected to deliver these goods to Location A in time. The transport is observed in real-time to detect any exceptional situations and thus respond immediately by setting corrective actions.

Figure 2.1 shows how the two maps interact. As for a new demand event, the best transport is selected in map "Storage Management", map "Transport Monitoring" observes this transport for exceptions, and collects metric data like carrier performance, transport costs and duration. These metrics are applied again in map "Storage Management" to select the best transport for the next demand event. So if a carrier causes more and more exceptions over time, it will be unlikely to receive the assignment for the next transport. In this way SENACTIVE InTime optimizes the business process by executing decisions always based on real-time information.

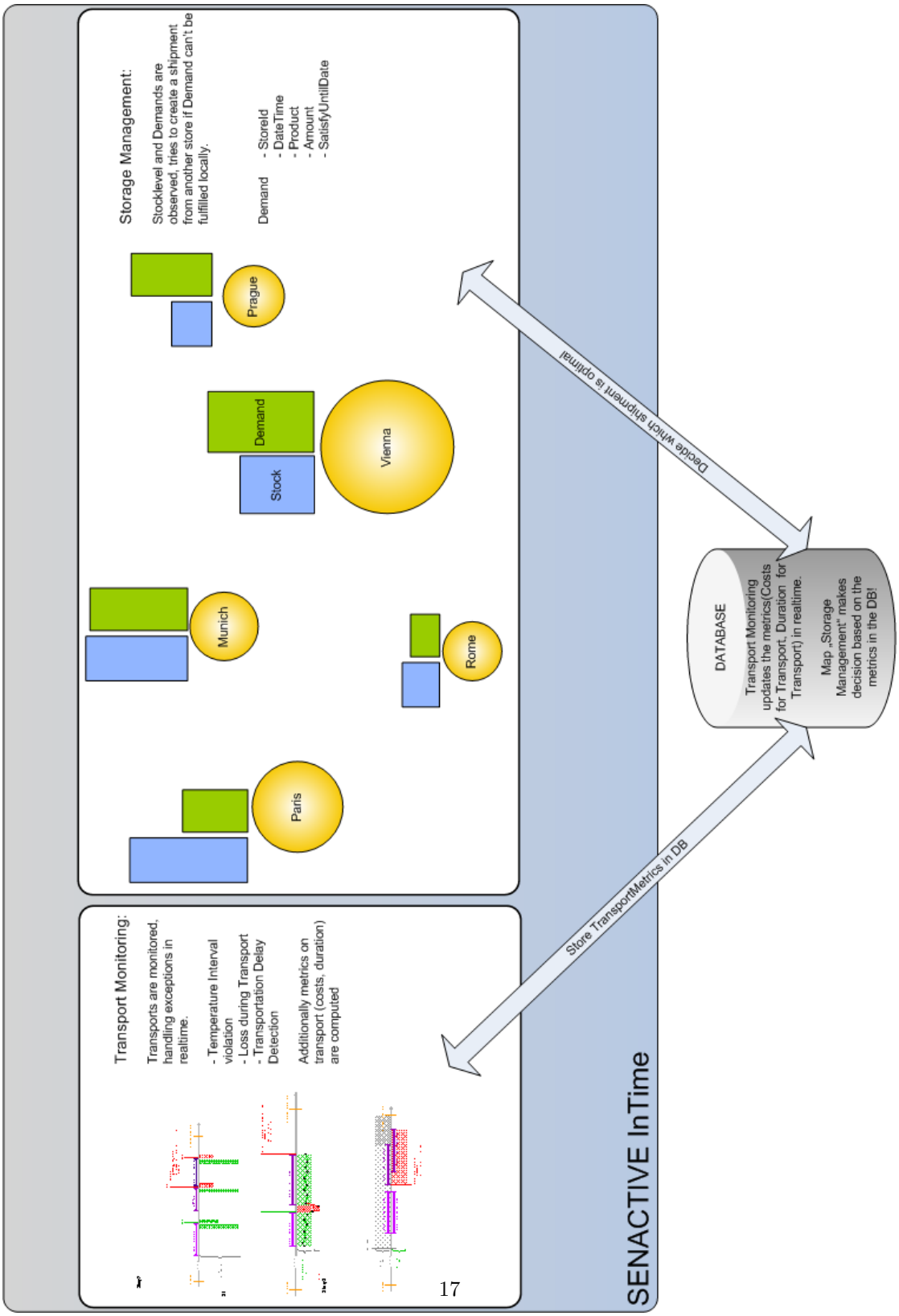

Figure 2.1: Medicare example - as implemented in SENACTIVE InTime

Figure 1.2 sketches a simplified version of the event processing map "Transport Monitoring". *TransportStart* and *TransportEnd* events are received as *input events*[1]. For every pair of *TransportStart-TransportEnd* three conditions are checked:

1. Was the temperature of the transported goods within the allowed interval or not (*Temp Violation Checker*)?
2. Was there a loss detected during the transport (*Loss Detection*)?
3. Was the transport delayed (*Transport Delay Detection*)?

Each of these conditions creates an *internal event* that propagates the state of the transport. The *Alert Rules* have the domain knowledge on how to proceed with the various conditions. For example if the transport is delayed then the transport may be told to accelerate, a second transport which would be faster could be triggered, or the domain rules could determine that taking no additional action is best. The *Alert Publisher* publishes the decision made, triggering actions in the real-world (*output events*).

2.2 Analyzing Medicare

2.2.1 Possibilities of OLAP

Once SENACTIVE InTime is set up for the Medicare Example, it runs autonomous without any direct user interaction. Therefore it does not include a graphical frontend for its calculated results and decisions per se. While a company's management is primarily interested in the additional profit the system generates, analysts and people in charge desire a deeper understanding of the events, the business situations and the current performance of the business that is supported by the CEP solution. To fulfill this wish, a graphical OLAP frontend was created to gain more insight.

Not shown in Figure 1.2 are additional steps that calculate metrics for the observed transports. To allow offline analysis of these transports, a mapping of the

[1] simplified assumption: additionally we would have more events like **DataLoggerRead**(for temperature violation), **ShipmentCreated** etc. because events will only represent a single activity

processed events into a relational star schema in a data warehouse(DWH) was created. Figure 2.2 shows the definition of this schema. The fact table includes transport information along with flags for the exception states (*LossInt*, *TempViolationInt*, DelayedInt) detected by SENACTIVE InTime. *Locations*, *Carriers* and *Date* are represented as dimensions.

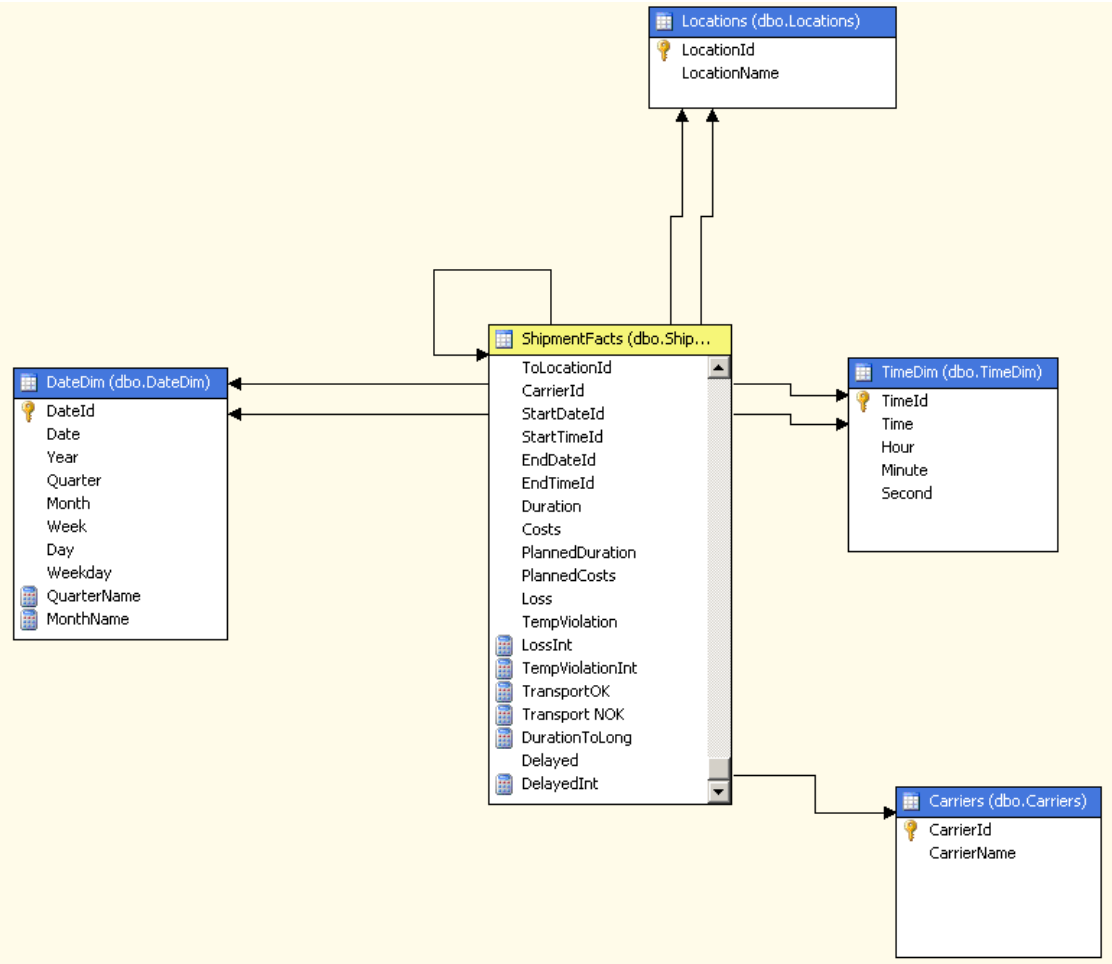

Figure 2.2: OLAP data cube for "Transport Monitoring" [9]

Using OLAP and a user-friendly frontend like Proclarity[2] allows analysts to query the transports monitored by SENACTIVE InTime as shown in Figure 2.3. OLAP

[2]http://www.proclarity.com/

allows to slice and dice the data cube to answer complex analytical questions. In Figure 2.3 we see the result of a simple query which selects the *effective* and the *planned duration* of all transports in the year 2006. By restricting dimension values and making a drill-down in date hierarchies, the query could easily be modified to "*Select effective duration, planned duration of all transports to London by carrier 'MegaTrans' in October 2006*".

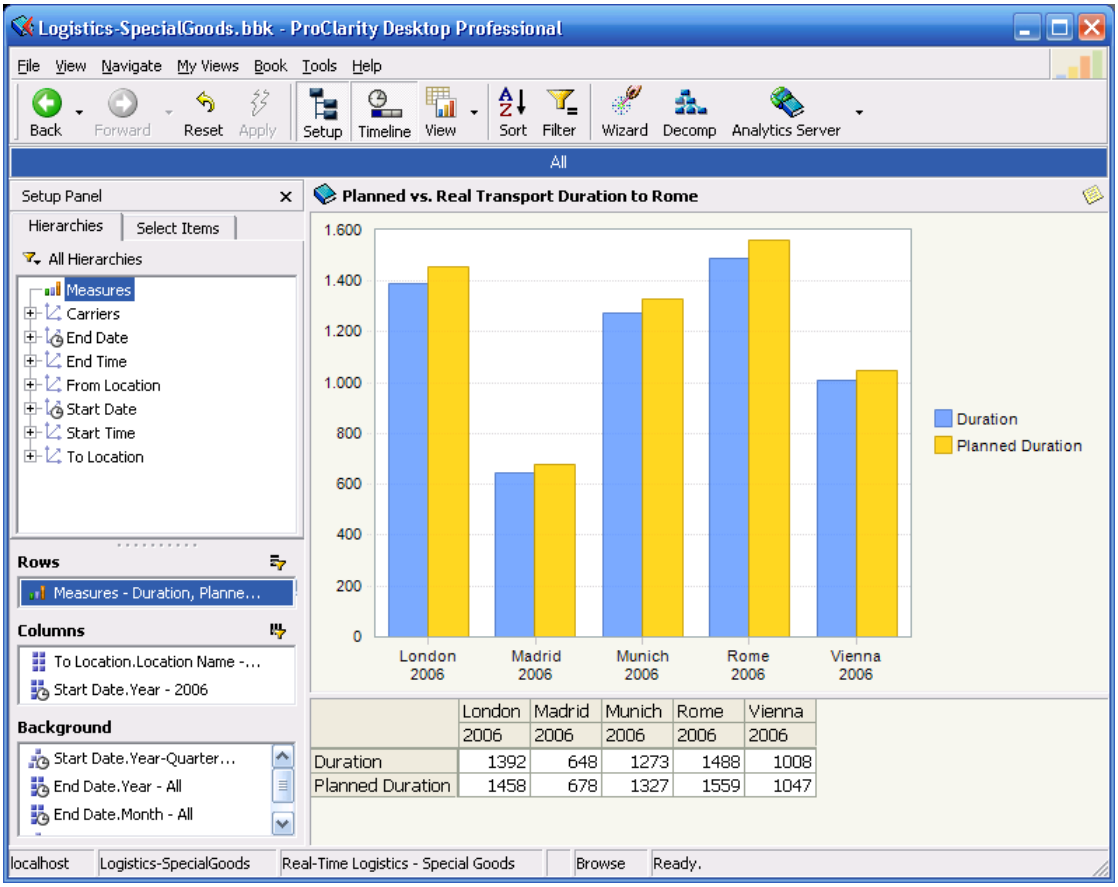

Figure 2.3: OLAP query in data cube [9]

2.2.2 Shortcomings OLAP - Need for Event Analysis

Even though OLAP offers a superb facility to query high-dimensional data, the reference to events has completely disappeared. DWH are data-centric, while CEP

solutions handle the events of business processes. By mapping the events of a CEP application into a relational data schema, information about events and their correlations is lost. That constitutes a breach between the world of event processing and the world of classical data representation. The requirements described in Section 1.3 for an event analysis tool are not fulfilled by the DWH, because the events and their correlations are not viewable by the user. Pattern search is not possible because the events are split into the fact table and multiple dimensions. The process steps represented by events such as *TransportStart*, *TransportEnd* are not available in the DWH. Just the overall outcome of an transport is stored as an entry in the fact table.

Nicholls points out these problems in [24]:

> *"The challenge here [...] is that data warehouses are data centric, storing data ready for analysis and reporting, not aligned to the process at hand.[...] Moreover, since the data warehouse often does not hold process-state data, let alone real time process state information, it usually does not have the information you need to make a decision."*, Nicholls [24]

Golfarelli, Rizzi and Cella reason that DWH is appropriate to higher level decisions and strategies, while lower level decisions need other technologies [10].

> *"[...] we might say that DWH is used by the top management to understand the enterprise and to define the global strategy, while other techniques must be used by tactical and operational decision-makers to "absorb" the strategy and make the best decisions for their tasks."*, Golfarelli, Rizzi, Cella [10]

In this example, questions similar to the following ones can be answered with OLAP and show the strength of OLAP at querying multidimensional data:

- How many transports have been delayed in 2006?
- Who is the cheapest carrier on the route Rome-Paris?
- What are the average transport costs, the average transport duration per route?

- What are the average transport costs with carrier "MegaTrans" on route Vienna-London within the last 24 hours?
- How does the overall business perform according to the available metrics?

Other analytical questions can'T be easily answered with OLAP queries, especially questions which strive to understand the incidences in a business process as well as inside a CEP solution. Examples would be to gain information about the process of decision-finding, to investigate in a specific transport, or to learn about the reasons for exceptions. This information is contained in the input events of the business process and in the internal and output events of the CEP solution, but partially lost when events are mapped into the relational data representation.

2.2.3 Analyzing Events

Questions will arise that can neither be answered with the CEP solution directly, as it does not offer facilities for analyzing its behavior, nor with a DWH. An analysis tool is needed that directly acts on (correlated) events so all information is available in it's natural representation. The following questions typically can be answered with EventCloud:

- **Which events and causal chains are related to a certain event?** Which events are related to event *TransportStart T2000*?
- **Which event(s) triggered a certain decision? What were the input events for decision-finding?** Why was Transport T2000 accelerated? (Because the internal *DelayDetected*-event informed about a delay of 6 hours).
- **I know there were process errors on a business case X! What has happened there?** Why was transport T2000 delayed? (Because the event *TransportStart* signalizes that the transport has left 8 hours to late).
- **What is the value of a metric for a certain business case?** What was the transport duration for transport T2000?

In Chapter 4, an alternative data representation for events is presented which is the basis for event analysis in EventCloud. It is a document oriented approach

following the ideas from Information Retrieval and utilizing the concept of event correlation to enable complex queries and analysis in events. Events can be directly stored in this data schema to avoid loss of information. With this data structure, it's possible to fulfill the requirements described in Section 1.3. Nevertheless OLAP, through its data representation, allows complex query mechanisms on multidimensional data of which EventCloud is not capable. As we will see in Chapter 7 again event correlation is used to implement an alternative approach to provide metric data in the EventCloud system.

While OLAP requires mapping the event data into a relational representation, EventCloud executes an additional processing step to add events to its data representation. The EventCloud system receives events and applies event correlation and metric calculation on them to create its data basis for event analysis. In opposite to OLAP, this "loading process" is from the very beginning a real-time approach. EventCloud follows an event-driven architecture adding events in near real-time to the data basis.

This preparation step is independent from the event processing in the CEP software. For analyzing events, other settings can be relevant than settings applied for real-time decision making of event streams done by the CEP application. Because of that event correlation and metric calculation for EventCloud is detached from the configuration in CEP even if the same events are processed. For example, in the analysis it is correct to correlate all events that happen for a specific order (*Demand, ShipmentCreated, TransportStart, TransportEnd, ShipmentReceived*) to make the complete order easily traceable, but for a CEP application one would only correlate *TransportStart* with *TransportEnd* to check for a transport delay.

2.2.4 Event Analysis without Complex Event Processing

Even this example suggests that event analysis is only meaningful if a CEP application is installed, but that is not always the case. In simpler scenarios, the business events are not the input for a CEP application but are the direct input for EventCloud. If no real-time 24*7 event processing is necessary, EventCloud

can be an accurate tool to analyze complex event scenarios. EventCloud includes the necessary facilities to exist independent from a CEP solution. For a user of the frontend, there is no difference if the data is processed directly or received from a CEP.

2.3 Summary

The first two chapters show the importance of events in today's business. Emerging technologies, like CEP, utilize these events to monitor and steer business processes. In this chapter we have seen how a use case can be implemented with a CEP software, bringing the need for a new generation of analysis tools which handle specifically business events.

Data warehouses are not sufficient to support analysis of business events as important information gets lost when events are mapped into a relational schema. To fulfill the requirements proposed by Luckham [22] other data representations must be found which focus on events. Through the rest of this thesis I am concerned with the concepts to create the event analysis tool EventCloud to fulfill these requirements. Chapter will pick up the Medicare example of this chapter to outline the facilities EventCloud provides for event analysis.

2.4 Definition EventServer vs. EventCloud

To avoid confusion I would like to define two terms that are repeatedly used throughout the rest of my thesis. The terms *EventServer* and *EventCloud* are used to describe the infrastructure for the proposed event analysis application.

As will be described later, the EventCloud relies on a CEP platform to create its data representation. This platform is called EventServer and provides real-time event processing, connectivity to other IT-systems, and system services like event correlation. Based on this platform, EventCloud implements its own services to process the events.

When writing about EventServer, it means exactly the server platform for event processing. However, the usage of EventCloud depends on the context: **first**, EventCloud is a synonym for "event analysis tools" and the idea of these tools, **second**, EventCloud sometimes means the user frontend for analyzing events, and **third**, EventCloud stands for the whole application stack including EventServer, EventCloud's backend functionality on top of the EventServer, and the user frontend.

Hopefully, with this definition, it's always clear from the context which part of the described system is being referred to.

3 An Approach for Event Correlation

One of the main concepts of EventCloud is the concept of event correlation. It is concerned with setting multiple events that occur at different points in time, which are of different types, representing different activities into a meaningful relationship.

Event Correlation is the basis for many topics in Complex Event Processing (CEP). The following chapters will illustrate that EventCloud depends on correlation when building the data representation and the search indexes for the processed events. Many other topics in event processing, like pattern detection and rule engines, can be realized on top of the concept of event correlation. For this reason, EventServer implements event correlation as a system service, available for use by application other than EventCloud.

The next chapter will give a detailed description about the concept of event correlation as implemented in the EventCloud system.

3.1 Motivation

With event processing there is always a challenge to get the context for a single event. While a single event contains almost no information besides the activity it signalizes, an event is normally just a small part of a complex chain of events representing a bigger activity. To have an added value from processing events, it's necessary to create these chains of events by applying correlation.

3.1.1 Correlation in a simple Network Protocol

Similar to the network example in [22] I would like to define a simple network protocol. This example helps to understand the necessity of bringing events which occur in a large stream of events into relation.

The network protocol has two message types which can interpreted as two types of events: *MESSAGEREQUEST(msgId, DateTime, From, To, Resource)* and *MESSAGESENT(msgId, DateTime, From, To, Data)*. *MESSAGEREQUEST* represents the event that a client has requested a service. *MESSAGESENT* represents the event that a server has responded a service request. For simplicity *MESSAGEREQUEST* and *MESSAGESENT* are the only two event types needed in this simple network protocol example. The entries in the brackets are event attributes, specifying values for the event type occurrence.

Imagine an event *MESSAGESENT(msgID=456, DateTime=2006-05-05T15:00:00, From=192.168.0.1, To=192.168.0.2, Data=htmlpage)* occurs. This event does not happen without reason. Servers do not send messages if nobody ask for them. Normally there would have been a matching request by some client, but from the point of view of the single event *MESSAGESENT*, you could not make any proposition about that. The event *MESSAGESENT* says nothing about other things that happened before. Maybe this message is sent to synchronize with other servers, maybe it's just an alive message. We can not know if we just see the single event.

An event processing system needs more than just events. It needs a possibility to connect, to relate multiple events to give them more meaning. That's where correlation comes into play.

Take a look at the *MESSAGEREQUEST* and *MESSAGESENT* events again. One typical information that is interesting in a network is the *Response Time*, i.e. how long did it take until the response for a request arrives. That is an interesting number but we would not expect that this information is sent along with an event of a network protocol. It is not a server's task to calculate this met-

ric, so there is probably no attribute for this metric in the *MESSAGESENT* event.

We need to calculate the *Response Time* on our own. Luckily that's an easy calculation:

Listing 3.1: Pseudocode Response Time Calculation

```
1 MESSAGESENT.DateTime - MESSAGEREQUEST.DateTime where MESSAGEREQUEST.msgId = MESSAGESENT.msgId
```

For an event processing system, however, it is not as trivial as it seems. Because event-driven systems like the EventServer always process a single event alone, the information about other events is not available per se. To solve this simple calculation, we need to correlate the two event types via their common attribute *msgId* and to store this correlation information in some data container, in a "session". Doing so allows us to calculate a value as shown in Listing 3.2. A pretty slow network, but the goal to generate additional knowledge a single event does not include is fulfilled.

Listing 3.2: Correlate MESSAGEREQUEST and MESSAGESENT to calculate the metric *Response Time*

```
1  MESSAGEREQUEST
2     msgId=456,
3     DateTime="2006-05-05T15:00:00"
4     From=192.168.0.2,
5     To=192.168.0.2,
6     Resource=index.html
7
8  MESSAGESENT
9     msgID=456,
10    DateTime="2006-05-05T15:00:05",
11    From=192.168.0.1,
12    To=192.168.0.2,
13    Data=htmlpage
14
15    ==> "ResponseTime" = 5sek
```

Correlating these two events with *msgId* is not the only possible correlation. A correlation is not tied to a unique identifier like *msgId*. Imagine another example where a history of all *MESSAGEREQUEST* events from a specific client are collected. In this case a correlation similar to Listing 3.3 needs to be defined.

Listing 3.3: Pseudocode: Requests by "Same Client" Correlation

```
1 MESSAGEREQUEST WITH SAME VALUE for MESSAGEREQUEST.From
```

Listing 3.4: ”Same Client 192.168.0.2” Correlation

```
1  MESSAGEREQUEST
2     msgID=456,
3     DateTime="2006-05-05T15:00:05",
4     From=192.168.0.2,
5     To=192.168.0.1,
6     Data=htmlpage
7
8  MESSAGEREQUEST
9     msgID=567,
10    DateTime="2006-05-05T15:00:12",
11    From=192.168.0.2,
12    To=192.168.0.1,
13    Data=otherpage
14
15 MESSAGEREQUEST
16    msgID=678,
17    DateTime="2006-05-05T15:00:33",
18    From=192.168.0.2,
19    To=192.168.0.25,
20    Data=htmlpage
```

Listing 3.4 shows that the client 192.168.0.2 sent two requests to the server 192.168.0.1, and later made a request to the server 192.168.0.25.

A correlation can be defined between any number of event types (from a single event type to any number of event types), on any number of attributes which must not be a unique identifier for an event type. An event type can participate on any number of correlations. Correlation is executed on the stream of processed events. While lots of events are processed, correlation is a particular view on these events only picking the events relevant for the correlation. Using a database metaphor, *correlation **selects** those events from a large stream of events that fulfill the given correlation criteria.*

Figure 3.1 shows how correlation can “select” events from a large stream of events. Independent from when the events occur, they are brought together by correlation. Correlated events share a common correlation session to exchange data or to calculate additional information like the metric *Response Time*. Events that are not correlated to each other have no facility to “communicate”.

3.1.2 Importance for EventCloud

But why is this all important for EventCloud? EventCloud is about searching and exploring events. One example was already presented: EventCloud will use

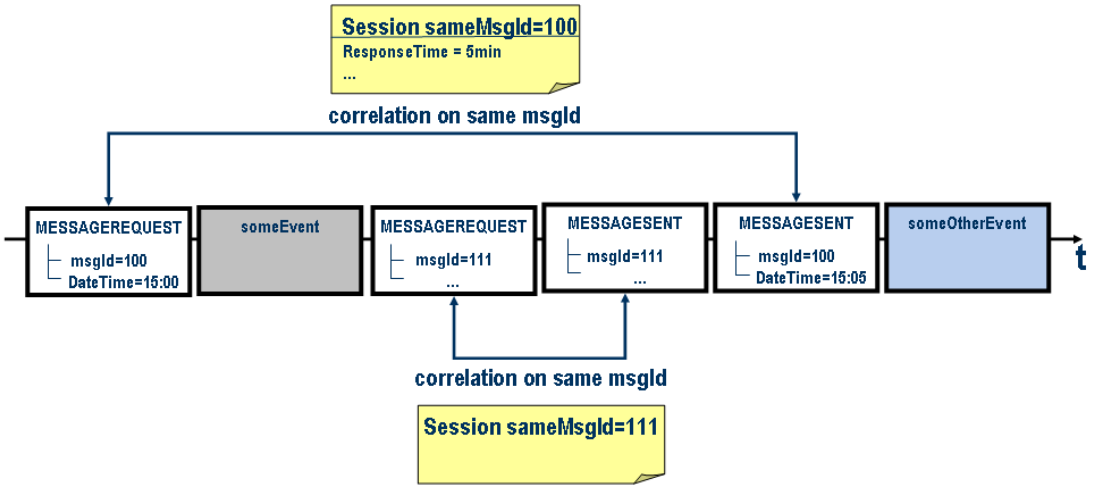

Figure 3.1: Select "events of interest" within an event stream with the help of correlation

correlation to calculate metrics like *Response Time*. But EventCloud, as we will see in the following chapters, also relies heavily on correlation to create an event search index which allows a user to answer complex search and analysis queries with EventCloud.

Therefore, I'd like to follow the Medicare example we already saw in Chapter 2. Figure 3.2 shows the four event types, *Demand*, *ShipmentCreated*, *TransportStart* and *TransportEnd*, that will be used throughout the rest of the thesis for examples of correlation. These four events are correlated via an indirect correlation ***AllOrderInformation***, spanning over all four event types. To realize this correlation, three conventional correlationsets have been defined: *demandToShipment*, *shipmentInfo* and *transportInfo*.

What's the idea behind this correlation? Why are these events correlated? In the Medicare example *Demand* events arise if some stocks do not have enough items to fulfill a local demand. Other stocks however may have enough items in their store to satisfy this demand. The *ShipmentCreated* event signalizes the decision that a transport from stock location A should deliver items to stock loca-

tion B to balance the demand. The *TransportEnd* and the *TransportStart* events represent facts about the actual transport triggered by the *ShipmentCreated* event.

We see that the correlation represents a complete business case, including all events of an order, from the *Demand* event to the actual delivery at *TransportEnd*. Direct correlation can not be applied in this case because no attribute of the *Demand* event is related to any attribute in the *Transport* events. Nevertheless, from a business point of view these events are conjuncted. Fortunately, we can use *ShipmentCreated* to bridge the gap and correlate all four events. Indirect (or "bridged") correlation helps to represent these business cases in a single correlation instance (more in Section 3.2).

Figure 3.2 "Definition" shows the event attributes available with each event type and how these four types are correlated. *Figure 3.2 "Runtime"* shows instances for each event type[1] with set values and materialize event correlations. The bridged correlation "AllOrderInformation" is spanned over all elements of the figure. In a real-world scenario, there would be hundreds of thousands instances of the correlation "AllOrderInformation" as every new *Demand* event would create a new instance of the correlation representing an individual business case.

For EventCloud, we want to exemplarily execute the following three search queries against the event full-text index. These are simple queries an analyst may execute against the four event types from Figure 3.2 defined in our example. I will demonstrate that a single event can't meet all of the queries. In order to return meaningful results to the user, I will explain how EventCloud uses event correlation as a concept for event search.

- "Madrid"
- "Madrid and Vienna"
- "ProductA and MegaTrans and Acceptor=Jane Smith"

[1] simply called an "event"

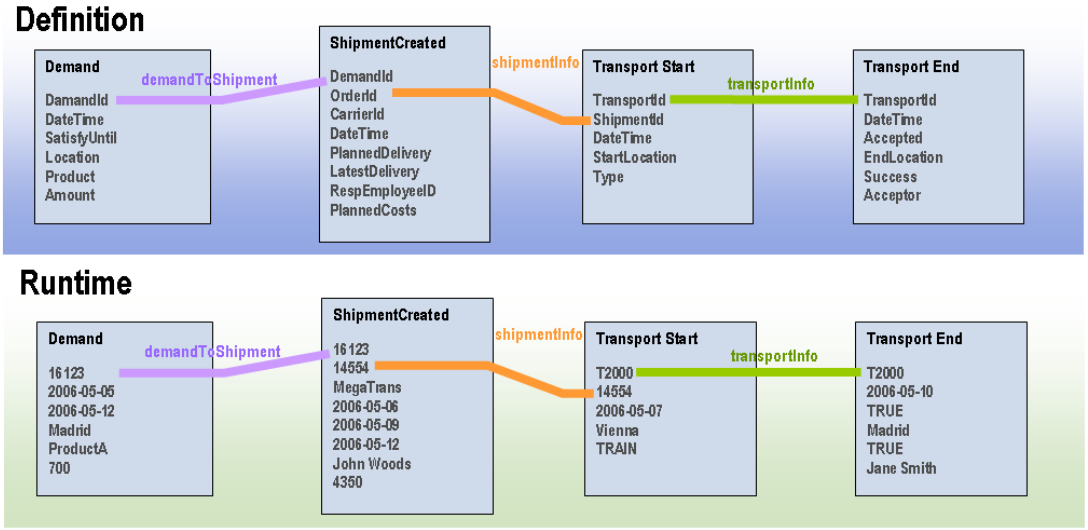

Figure 3.2: Event types correlated via the bridged correlation *AllOrderInformation* consisting of the correlationsets *demandToShipment*, *shipmentInfo* and *transportInfo*

We will see that EventCloud depends on correlation to get results for the more complex queries. But before going into these event search examples, the next section describes the nomenclature used for event correlation in EventCloud.

3.2 Terms and Definitions

For the rest of the text, a specific nomenclature for the concepts in event processing needs to be defined. It's necessary to understand the terms to follow the upcoming chapters. The nomenclature is, in most parts, taken from SENACTIVE InTime [9] as they implement similar concepts.

The definitions of correlation work on top of events. It is necessary to first define exactly what an event is and what parts an event contains.

3.2.1 Event

An event represents a notable activity which just happened. Depending on the time granularity needed, an activity is signalized by a single event (like *MES-*

SAGEREQUEST) or by multiple events if the activity lasts for some time (like *TransportStart*, *TransportEnd*). An event carries information illustrating the activity it represents in its *event attributes*. *Event attributes* are also used to correlate an event with other events. A related definition can be found at [22] and [23].

3.2.2 Event Type

An event type defines how a particular type of event looks like. It's similar to the concept of a *Class* in object oriented languages. During development, event types of the events that should be represented in the system need to be defined. During runtime, instances of the defined event types are created which are then called *events*.

An event type includes minimally

- A unique name.
- A list of attributes this event stores for the activity it represents.

SENACTIVE InTime includes some extensions on event types that are interesting to mention:

AllowUnknownAttributes a weak typed events, that can include any additional attribute at runtime.

Parent Event types similar to object orientation inheritance, it's possible to extend event types.

Virtual Event types the attributes of the event types defining the virtual event type are conjuncted. The virtual event has only attributes that are available in all event types.

EventCloud only uses a unique name and the definition of the event attributes. But we see that an elaborate type system can be interesting for complex event processing. An extended type concept might be interesting in a future version of EventCloud and can be the area of new research.

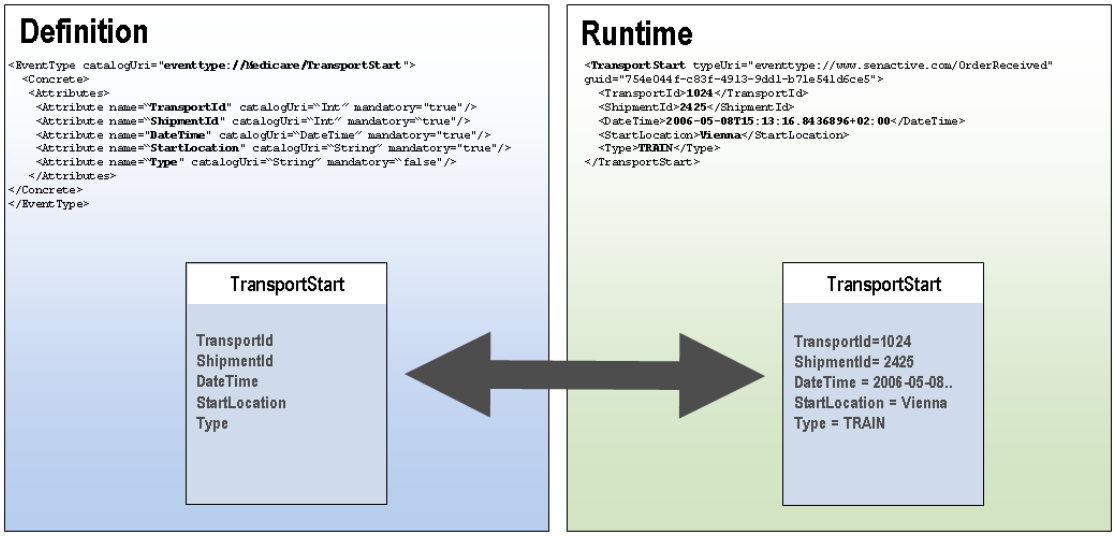

Figure 3.3: EventType declaration and an event at runtime

Figure 3.3 shows the XML-definition of the event type *eventtype://Medicare/TransportStart*, having five attributes. In this example, the event attributes are strongly typed as known from programming languages like Java. An attribute can be mandatory or not. At runtime, an event can be seen as an instance of the event type. The attribute fields are "instance variables" of the event. In this example, we see the event that represents the event type *TransportStart* with *TransportId=1024*. An event is uniquely defined by a GUID in the events header not shown in 3.3. All event attributes may be implicit identifiers for this event, but depending on the scenario an event generated in it may not include any event attribute identifier.

3.2.3 Event Correlation: Correlationset, Correlation Instance and Correlation Session

The previous section describes the definition of event types. Correlations are also defined declaratively, on top of the concept of event types. We distinguish between two types of correlation: the definition of a correlation between event types is called a *Correlationset*. A correlationset **directly** correlates event types. If we put multiple *correlationsets* into a relationship it is a *Bridged Correlation*. Bridged

Correlation allows *indirect* correlation of event types.

At runtime, multiple instances of a correlation exist. We call them *Correlation Instances*. As a correlationset defines which attribute values of the event types have to match to relate events, a *Correlation Instance* represents the correlation for a specific attribute value. Each *Correlation Instance* is backed up by a data container called *correlation session*.

Correlation session was defined by Schiefer and McGregor in [29] as

> *"A container with a set of data items that exists for each relationship between events [...] we define a correlation session CS as a triple of the form (O, X, A) where O defines the owner, X the correlation expression and A the correlating event attributes for the session."*, Schiefer and McGregor [29]

Correlation sessions are comparable to the concept known from HTTP sessions. Every HTTP request of a user can be seen as an event. Relevant data which is needed to fulfill later requests (events) is stored in the session. Each further request of the same user will activate the same session. Simplified, we can say that in a webserver the relation between multiple requests is defined with the correlation "*same User*". For each user, an own session exists and every request by a user X always activates the same session Y.

If user X sends a further request, the webserver activates the session depending on the session identifier sent along with the user request. This example is similar to what we have seen in Listing 3.4 where the event MESSAGEREQUEST was correlated over the event attribute *msgId*.

Following the definition of Schiefer and McGregor[29], a web session can be interpreted as correlation session by:

"O" ... The webserver is the (only) owner of the session. No one else is allowed to utilize the session

"X" ... The attribute where to find the session information. Normally this is the key *Cookie:* in the HTTP header, or a specific parameter like *SessionID* encoded in the URL

"A" ... the value of the key *Cookie:*, or the actual value of the URL-parameter. This value points to a specific session.

However, the concept of event correlation defined by [29] provides a broader approach and offers arbitrary possibilities to define correlations between event types. Correlationsets give a developer free hand on which event types to correlate and how, because a meaningful correlation definition always depends on the concrete scenario a correlation is applied to. As the web sessions is just one use case for correlation, any number of other scenarios are imaginable. EventCloud implements event correlation to be applicable to any domain.

A correlationset consists of:

- A unique name
- The event types that participate in this correlationset
- The event attributes which have to match

Listing 3.5: Definition of a correlationset for a webserver's HTTP session

```
<Correlationset identifier="WebserverSession">
  <CorrelationTuple>
    <CorrelationData eventtypeUri="eventtype://MESSAGEREQUEST">
      <XPathSelector>//From</XPathSelector>
    </CorrelationData>
  </CorrelationTuple>
</Correlationset>
```

Listing 3.6: Definition of a correlationset with multiple event types

```
<Correlationset identifier="transportInfo">
  <CorrelationTuple>
    <CorrelationData eventtypeUri="eventtype://MediCare/TransportStart">
      <XPathSelector>//TransportId</XPathSelector>
    </CorrelationData>
    <CorrelationData eventtypeUri="eventtype://MediCare/TransportEnd">
      <XPathSelector>//TransportId</XPathSelector>
    </CorrelationData>
  </CorrelationTuple>
</Correlationset>
```

Listing 3.5 and Listing 3.6 show two simple examples of correlationsets. For every event type that should be correlated in the correlationset an entry *<CorrelationData>* has to be defined. The tag <XPathSelector> includes the XPath expression evaluated on the event to match a correlation session. Currently all correlationdatas are compared with an *EQUALS*-operator in EventCloud. This is sufficient for many cases, however the concept of correlationset is extensible to other correlation operators as well.

At runtime, the values selected by the *XPathSelector(s)* from an event are called *correlation value*. A correlation value points to exactly one correlation session, so a correlation value is used to retrieve the correct session for a correlated event. For an event *TransportStart* with an event attribute *TransportId=T2000*, according to Listing 3.6, the correlation session retrieved is uniquely identified by the correlation value *transportInfo=T2000*. Another event *TransportStart* with the event attribute *TransportId=T2001* is handled by another correlation instance that is backed-up by another correlation session identified by the correlation value *transportInfo=T2001*.

A correlationset can be interpreted as a template for the *eventstream of interest* during runtime. If an event is processed in EventCloud, the *correlation service* looks up which correlationsets the event type of the current event participates. Next it selects the *CorrelationData* from the event for the given correlationset. If a session for this value exists, the session for this correlation instance is returned. Otherwise a new session is created and then returned.

Figure 3.4 shows how sessions are found through the correlation values selected from the event attribute. The correlationset with identifier *transportInfo* is defined in the system as in Listing 3.6. The event *TransportStart* is processed by the system. At **1)** the correlation value *transportInfo=T2000* is selected because of the correlationset definition. At **2)** the correlation service checks if a session for the correlationset *transportInfo* with the value *"T2000"* already exists. If so, the session is returned, otherwise a new session is created and returned.

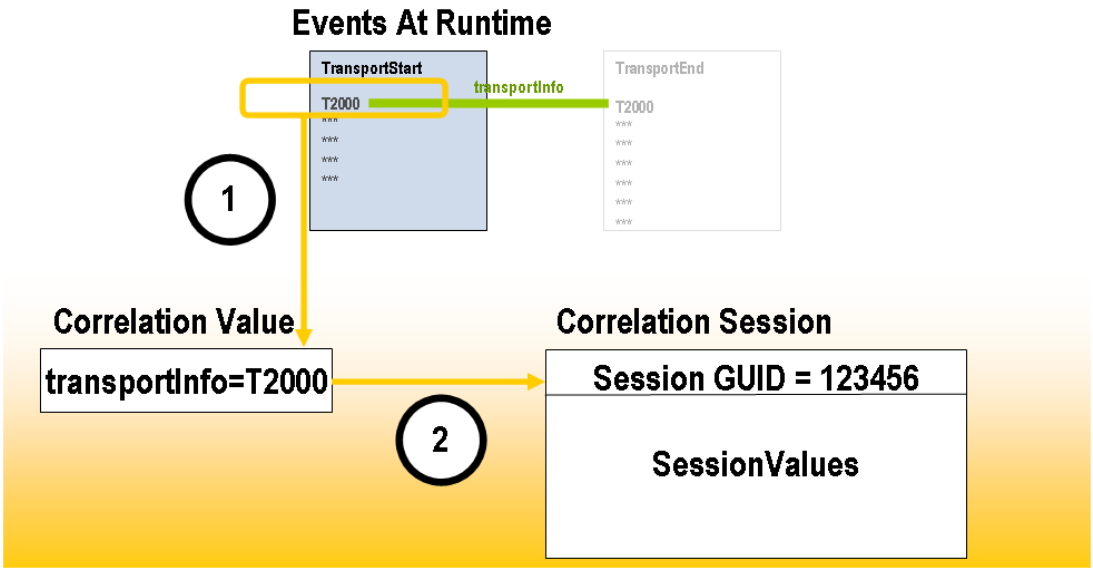

Figure 3.4: correlation value identifies correlation session at runtime

Later, an event *TransportEnd* with *TransportId=T2000* is processed, and the correlation service will return the same correlation session object with *Session GUID=123456* as before. Since a session is defined as a data container object, the session allows an exchange of information across events. We already saw the implementation of a simple duration metric with the help of correlation, and later chapters will show how EventCloud makes use of this concept.

3.2.4 Bridged Correlation

A Bridged Correlation defines an **indirect** correlation of event types, which could not be correlated otherwise. Bridged Correlations are one aggregation level higher than the correlationsets defined previously: Multiple correlationsets together represent a single bridged correlation.

The goal of a bridged correlation is to correlate events that cannot be correlated by a single correlationset. A bridged correlation was shown in Figure 3.2. The *Demand* and the *TransportStart* event type have no event attribute in common to correlate. However via *ShipmentCreated* an indirect correlation can be built with the help of the two correlationsets *demandToShipment* and *shipmentInfo*. If additionally the event type *TransportEnd* and the correlationset *transportInfo* are added to the bridged correlation, all four events can be *correlated*.

Bridged correlations are not automatically created from multiple correlationsets. Even this inference step would be possible via the event attributes, correlationsets, as well as bridged correlations, are currently configured by a user manually.

A bridged correlation is said to be *weaker* than a direct correlation. The *Demand* event is weaker correlated to the *TransportEnd* event than the *TransportStart* event is. This circumstance is also considered when we build the full-text search index for EventCloud.

A bridged correlation consists of:

- An unique name
- A list of correlationsets in the Bridged Correlation

Listing 3.7: Definition of a bridged correlation *AllOrderInformation* over multiple correlationsets

```
1 <BridgedCorrelation identifier="AllOrderInformation">
2   <Description>
3   Sample bridged correlation from the Medicare example
4   </Description>
5   <correlationset identifier="demandToShipment">
6     <CorrelationTuple>
```

```
7         <CorrelationData eventtypeUri="eventtype://MediCare/Demand">
8           <XPathSelector>//DemandId</XPathSelector>
9         </CorrelationData>
10        <CorrelationData eventtypeUri="eventtype://MediCare/ShipmentCreated">
11          <XPathSelector>//DemandId</XPathSelector>
12        </CorrelationData>
13      </CorrelationTuple>
14    </correlationset>
15    <correlationset identifier="shipmentInfo">
16      <CorrelationTuple>
17        <CorrelationData eventtypeUri="eventtype://MediCare/ShipmentCreated">
18          <XPathSelector>//OrderId</XPathSelector>
19        </CorrelationData>
20        <CorrelationData eventtypeUri="eventtype://MediCare/TransportStart">
21          <XPathSelector>//ShipmentId</XPathSelector>
22        </CorrelationData>
23      </CorrelationTuple>
24    </correlationset>
25    <correlationset identifier="transportInfo">
26      <CorrelationTuple>
27        <CorrelationData eventtypeUri="eventtype://MediCare/TransportStart">
28          <XPathSelector>//TransportId</XPathSelector>
29        </CorrelationData>
30        <CorrelationData eventtypeUri="eventtype://MediCare/TransportEnd">
31          <XPathSelector>//TransportId</XPathSelector>
32        </CorrelationData>
33      </CorrelationTuple>
34    </correlationset>
35  </BridgedCorrelation>
```

The basic execution of a bridged correlation in EventCloud is very similar to a correlationset. Every bridged correlation returns a correlation session at runtime. An event activating a (bridged) correlation session always gets a single, independent session returned. Because of this characteristic, handling bridged correlations at runtime is more complex than handling simple correlationsets. This point will be discussed further in Section 3.3.

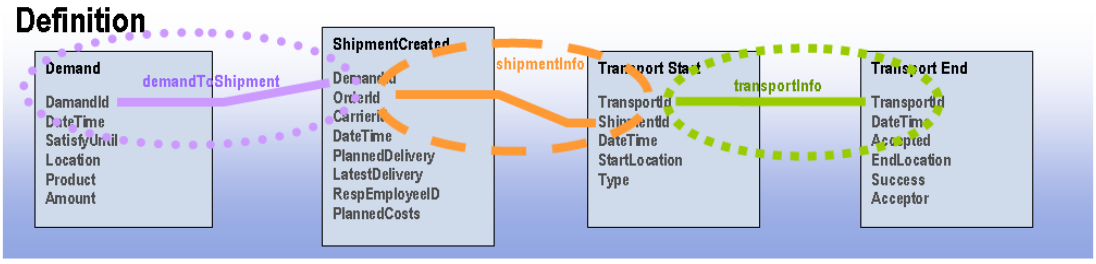

Figure 3.5: Three independent correlationsets *demandToShipment*, *shipmentInfo* and *transportInfo* defined (without bridged correlation)

Let's say the three correlationsets *demandToShipment*, *shipmentInfo* and *transportInfo* are defined without a bridged correlation. So at runtime, after processing all four events, three different correlation sessions will exist in EventCloud. Every

correlation session is indicated by a cycle in Figure 3.5. These three correlation sessions are independent from one another. The *Demand* event is in no way correlated with the *TransportEnd* event. **Exactly one correlation value points to every session**.

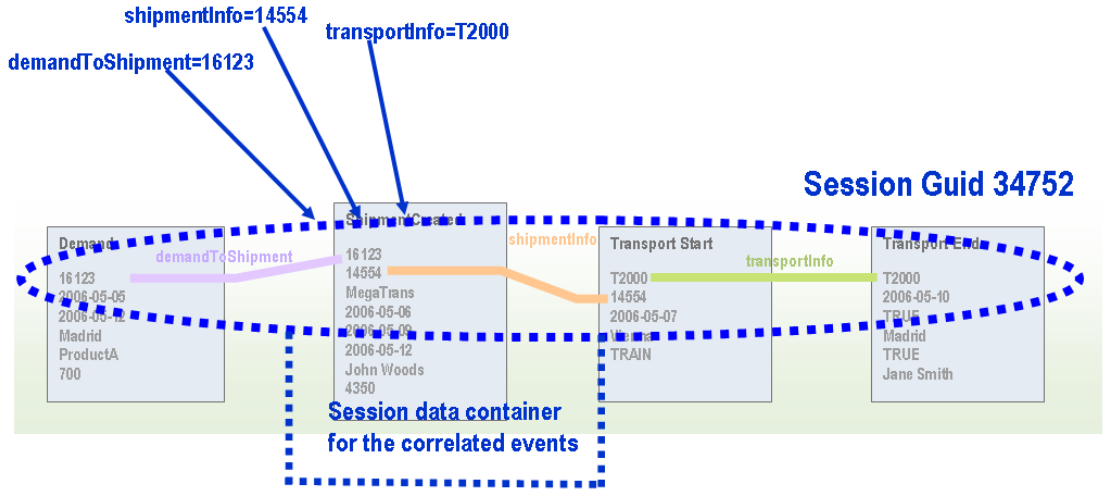

Figure 3.6: Bridged Correlation causes a single session for all four events

We now define a bridged correlation over the three independent correlationsets as shown in listing 3.7 above. Now, only a single correlation session for the four events would exist when they are processed. The correlationset *shipmentInfo* closes the gap, so it's possible to correlate events that have obviously something to do with each other[2], but could not be correlated directly. In contrast to simple correlationsets, in this case multiple correlation values point at the **bridged correlation session**. So different requests at the correlation service can lead to the same session. Figure 3.6 shows that a single correlation session spans over all four events so the session is the common data container for all four events.

[2]they all occur in the "Demand causes Transport" episode

3.3 Bridged Correlation at Runtime

In Section 3.2.4 I argued that a bridged correlation is represented by a single correlation session. That's not exactly true during execution. With a bridged correlation, a *Session Merge* situation which can occur during runtime must be transparently managed by the correlation service.

A *Session Merge* can be defined as:

> *When not all event types participating in a bridged correlation are already processed and events arrive in a particular order the bridging event could cause a Session Merge of two independently existing correlation sessions. This merge has to happen transparently for all requests because one event can always obtain only one single correlation session for a specific bridged correlation.*

We call an event *Bridging Event* when multiple correlationsets of a bridged correlation are defined on this event. In the bridged correlation in Figure 3.6 the events *ShipmentCreated* and *TransportStart* are bridging events. It can be seen that these two events allow to correlate the events *Demand* and *TransportEnd* which themselves cannot be correlated directly. So *ShipmentCreated* and *TransportStart* build a *bridge* between them.

ShipmentCreated is a bridging event because of the correlationsets *demandToShipment* and *shipmentInfo*.

TransportStart is a bridging event because of the correlationsets *shipmentInfo* and *transportInfo*.

The order in which the events are processed has an impact on how correlation needs to work in the background. Because events are received from different, external source systems, no order is guaranteed. If the bridging event is processed last, some event types of a bridged correlation cannot be correlated in the meantime. The bridging event is needed to realize the indirect correlation. Without the bridging event, parts of the bridged correlation can exist in independent correlation sessions. With the bridging event, these parts can be correlated causing

a *Session Merge*.

Four different situations can occur if a session is requested by an event:

Correlation Session does not exist An event causes a correlation session to be newly created. All event's correlation values are added to the session.

Simple Merge The correlation session already exists but not all correlation values of the event are handled by the session yet. Simply add the missing correlation values to the session. This case does not need a session merge. It simply extends the *"responsibility"* of a correlation session. As this occurs very frequently during runtime, it must be handled with high performance by the correlation service.

Complex Merge Different correlation sessions exists for a bridged correlation in parallel. This can happen when events occur in such an order that they cannot be correlated until the bridging event occurs. However when the bridging event occurs, these two sessions have to be merged into a single correlation session.

This situation is the most complex to handle, yet it does not happen very often. First, the events must unfortunately be processed in an order so the bridging event is processed late, and second, if a merge occurs once in a correlation session, it must not occur again.

Already Merged All merges of a bridged correlationset have already been processed. For all further requests on this session by later events, the correlation session can simply be returned.

Bridged Correlation, as well as normal event correlation, is a *commutative* operation. The order in which events are processed has no effects on the resulting correlation.

$$f(x, y) = f(y, x)$$

The function above shows this commutative operation, where x and y are events processed. The result of an event correlation f is a correlation session. The com-

mutativity could be expanded to any number of events in the correlation operation *f*.

3.3.1 Bridged Correlation Runtime Example

Again take a look at the example with the four event types correlated in the bridged correlation *"AllOrderInformation"*(Figure 3.2). Let's reproduce what happens if the events are processed in different orders. Be aware that the events are received by the EventServer from different other IT systems. First, the *Demand* event may be received from a statistical analysis software, then *TransportStart* and *TransportEnd* are sent to the system from a carrier's subsystem, and finally, *ShipmentCreated* is generated by a local transport planning software. As different systems have different configurations, performance, and delays, it's possible that events are not received and processed in their natural order. **Correlation must be aware of this problem and in the end has to produce the same result independent from the order in which events are processed**.

Simple Merge

The following sequence of events shows the default behavior of a bridged correlation. A single session expands it's scope until all event types of the bridged correlationset participate on the same session at least once.

1. **Demand with DemandId=16123** Session *AllOrderInformation* | *demandToShipment=16123* created
2. **ShipmentCreated with DemandId=16123, OrderId=14554** Session *AllOrderInformation* | *demandToShipment=16123* retrieved and expanded for correlation value *shipmentInfo=14554*: *AllOrderInformation* | *demandToShipment=16123* | *shipmentInfo=14554*
3. **TransportStart with OrderId=14554, TransportId=T2000** Session *AllOrderInformation* | *demandToShipment=16123* | *shipmentInfo=14554* retrieved and expanded for correlation value *transportId=T2000*: *AllOrderInformation* | *DemandToShipment=16123* | *shipmentInfo=14554* | *transportInfo=T2000*

4. TransportEnd with TransportId=T2000 Session *AllOrderInformation* | *demandToShipment=16123* | *shipmentInfo=14554* | *transportInfo=T2000* retrieved

During runtime, a single correlation session for the bridged correlation *AllOrderInformation* exists at all times. The session only extends it's scope (Simple Merge) as more correlated events are processed. At time **1)** it's just the correlation session for all *Demand* and *ShipmentCreated* events with *DemandId=16123*. When at time **2)** *ShipmentCreated* is processed the correlation session 456789 is expanded and also responsible for the correlationset *shipmentInfo* for *OrderId=14554*. At time **3)** the same happens for the correlationset *transportInfo* for *TransportId=T2000*. Figure 3.7 shows the situation when *TransportStart* is processed. From the correlation value *shipmentInfo=14554*, the event can already activate the session but for correlation value *transportInfo=T2000* no lookup exists. So to allow correlation with future *TransportEnd* events, the correlation value *transportInfo=T2000* needs to point to the session 456789 too. When *TransportEnd* is processed at **4)** it can obtain the session 456789 with correlation value *transportInfo=T2000* representing a fully occupied instance of the bridged correlation we have defined.

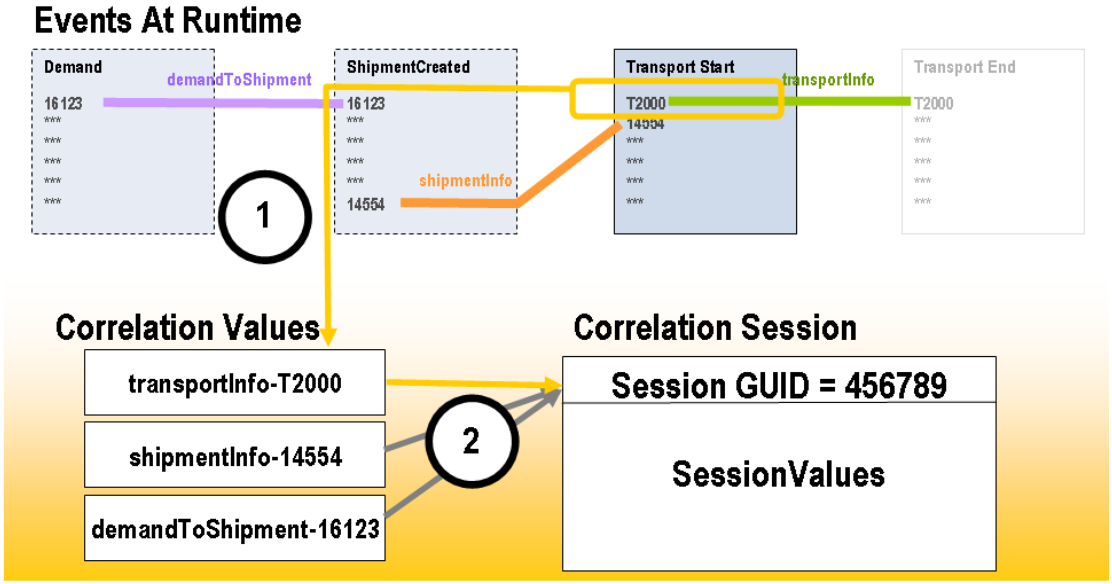

Figure 3.7: Simple Merge when *TransportStart* is processed. The correlation value *transportInfo=T2000* is added to the responsibility of session 456789

Complex Merge

The following event sequence shows the need of a *Session Merge* with bridged correlations. First a situation is described where two independent correlation sessions exists during runtime:

1. **Demand with DemandId=16123** Session *AllOrderInformation* | *demandToShipment=16123* with *sessionGUID=55555* created
2. **TransportStart with OrderId=14554, TransportId=T2000** Session *AllOrderInformation* | *shipmentInfo=14554* | *transportInfo=T2000* with *sessionGUID=65432* created
3. **TransportEnd with TransportId=T2000** Session *AllOrderInformation* | *shipmentInfo=14554* | *transportInfo=T2000* retrieved

At this point we have two sessions at runtime. They exist until the bridging event is processed. This is the situation shown in Figure 3.8. The *Demand* event has created its own session with the correlation value *demandToShipment=16123*. *TransportStart* and *TransportEnd* have created another session with the correlation values *shipmentInfo=14554* and *transportInfo=T2000*. The correlation service has been able to correlate the two transport events through their transport identifier, but until now it was not possible to correlate them with the *Demand* event. No assertion can be done if this *Demand* event is in anyway related to the two events in transport *T2000*. Any other *Demand* event could be correlated to transport *T2000* as well.

Figure 3.9 shows the *Session Merge* proceeding when the bridging event *ShipmentCreated* is processed in the fourth step. At **1)** the correlation values *demandToShipment=16123* and *shipmentInfo=14554* are extracted from the event attributes *DemandId* and *OrderId*. With these values, the correlation session is requested. As two sessions *55555* and *65432* are retrieved, a *Session Merge* is signalized at **2)**. A strategy decides which of the two session will survive and which session will be merged. At **3)** we see that the session *55555* "wins" the merge and from now on represents the bridged correlation session *AllOrderInformation*

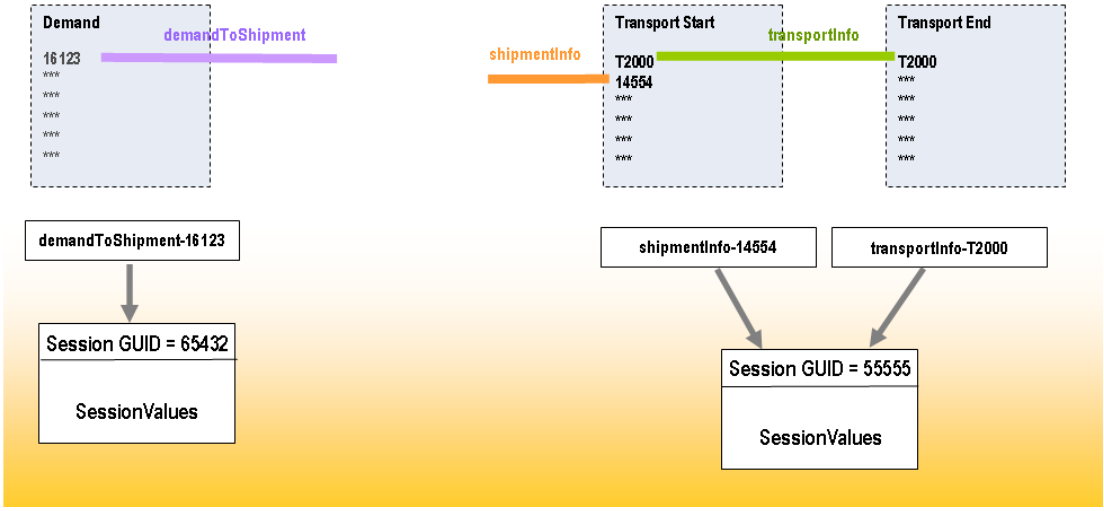

Figure 3.8: Bridged Correlation represented by two independent sessions, because bridging event *ShipmentCreated* is not processed yet.

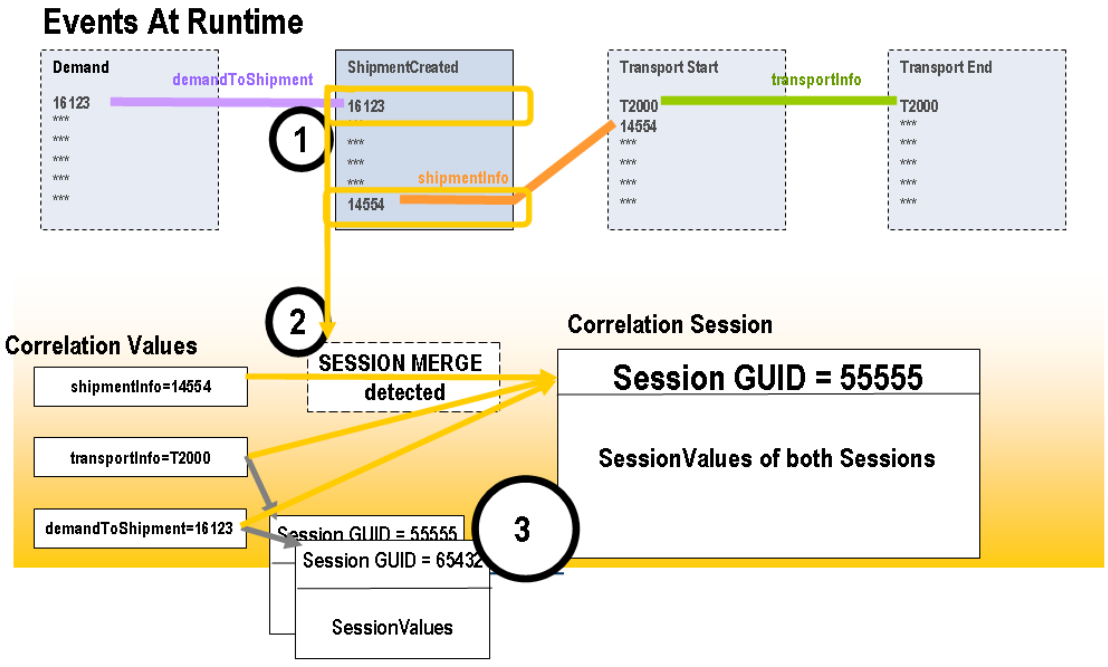

Figure 3.9: Bridged Correlation after processing the four events

| *demandToShipment=16123* | *shipmentInfo=14554* | *transportInfo=T2000*. The correlation values *shipmentInfo=14554* and *transportInfo=T2000* must be reassigned to the merge winner *session 55555*.

When the *ShipmentCreated* event is processed, the *Demand [DemandId=16123]* event can be ultimately connected with the *TransportStart* and *TransportEnd* event of transport *T2000* via the *ShipmentCreated [DemandId=16123, OrderId=14554]* event that creates a correlation bridge between these events.

All further requests with one or more of the three correlation values will always return session *55555*, because now all three correlation values point to the same session. When the "Session Merge" once happens, the bridged correlation behaves like every other correlation. A bridged correlation always collapse into a single correlation session finally.

This complexity arises because it's unknown in advance which correlation values the bridged correlation will take at runtime. If a correlation session for *demandToShipment=16123* and a correlation session for *transportInfo=T2000* exist, they are in no way related to each other. Until the matching event *ShipmentCreated* closes the bridge and allowing to recognize that the events (and thus their correlation sessions) "fulfill" the bridged correlation.

Merging the objects representing the correlation sessions is not a trivial task. Correlation sessions are data containers where a user can store any serializable data. Data is stored as a key-value pair. If two correlation sessions that are merged include same keys, a strategy is needed to solve this conflict.

In EventServer, *Hashtables* must be merged in the session objects. Because duplicated keys are not allowed in a *Hashtable*, the simplest behavior is to throw an exception if a key occurs in both sessions. However, that's just the simplest policy and not sufficient in most situations. For a correlation service, the strategy to merge session objects should be pluggable to allow executing different policies

depending on the requirements.

3.4 Related Work - Event Correlation and Event Processing

Early approaches for event correlation have been collected in [26]. Event correlation has its origins in network monitoring applications. Events can represent problems and/or symptoms of problems in a network. To solve the problem, the root cause must be found. [37] represents the causal relationships between problems and symptoms with a causality graph which is used to create codes for the set of events (symptoms) caused by a problem to identify this problem. Currently observed symptoms are decoded to determine which problem has the observed symptoms as its code. The approach is used to quickly correlate involved events to their root cause. [35] has *rule* and *action* types to correlate events and execute proper reactions. Commercial tools like HP OpenView are available in this area, but the concept of event correlation is always tied to network events.

However, CEP was introduced to be applicable to any domain. [29] points out the importance of event correlation in business monitoring. Event correlation allows to calculate business metrics on top of business process events in near real-time. This approach fundamentally differs from traditional DWH-concepts with a batch ETL-process.

The phrase *complex event processing(CEP)* was coined by David Luckham in [22] coming from Luckham's experience with the Rapide event pattern language [23]. On the other hand, there was significant research done in the area of *continuous queries*[2] now generating lots of momentum under the name *event stream processing(ESP)*. The research projects Aurora [6], Stanford STREAM [12] and Borealis [1] are noteworthy. Streambase [32] is a spin-off company of the Aurora project. Recently the first open source engine, Esper, was published [14].

The projects in the area of ESP implement event processing by executing *continuous queries* on event streams, similar to the concepts known from active databases

[25]. All programming logic on eventstreams is executed via user defined queries. The syntax of these queries is related to SQL statements and was published, for example, for the TelegraphCQ project [7]. Aurora as well as STREAM use SQL-like query languages. These query languages execute operations similar to SQL, including:

SELECT Selects event types, attributes of an event in the eventstream.

WHERE Define conditions for the events that should fulfill the query.

AGGREGATION Min, Max and other aggregations known from SQL are available.

JOIN Similar to our definition of event correlation, events can be joined via their attributes.

TIMEWINDOW The queries are executed against the events in a specific (sliding) timewindow.

Listing 3.8: Definition of the Response Time metric with Esper

```
1  select  MESSAGESENT.msgId as MessageId,
2          (MESSAGESENT.datetime - MESSAGEREQUESTED.datetime) as ResponseTime
3          from  MESSAGESENT,
4                MESSAGEREQUESTED
5  where MESSAGESENT.msgId = MESSAGEREQUESTED.msgId
```

Listing 3.9: Continuous queries for stock prices

```
1
2  select  avg(price) from StockTickEvent.win:time(300) where StockTickEvent.symbol='IBM'
3
4  select  symbol, avg(price) as averagePrice
5          from StockTickEvent.win:length(100)
6  group by symbol
```

The queries are executed at runtime on the stream of events for every event processed, so events (or event data) can be selected from the stream. The idea is to set on top of the well understood SQL language so people can easily adapt and reuse their skills. However, by now all these languages are proprietary and there is only some vendor driven effort to define a common standard [33].

Listing 3.8 shows how the calculation of the metric *ResponseTime* can be done

with Esper. The syntax is in this case similar to SQL and very intuitive if one is familiar with SQL. Even queries that use SQL-extensions are easily understood. The first query in Listing 3.9 returns the average price of all IBM stock tick event within the last 300 seconds (with a sliding time window). The second query returns the average price per symbol for the last 100 stock ticks.

Event correlation for the queries is done in the background: as the queries execute on time windows and connect multiple events via JOINs, the state of each query needs to be persisted in the background. This is a very similar concept to the correlation session defined in Section 3.2.3, except that the session is actively used as data container, whereas state-handling for continuous queries is done in the background.

The approach I present in my thesis was published in [29] and follows a different concept: instead of doing everything with a SQL-like query language, event correlation is used to select related events. These correlated events share a common session container and can be processed by any program code allowing much more flexibility and expressiveness than SQL-like queries. The concept of event services will be presented in Chapter 5. An event service can use any programming language construct to execute its task. This is not quite possible with continuous queries. The event services that create the EventCloud data representation, and therefore depend on the Apache Lucene API to create a full-text search index, are an example for the need of event services in complex event processing.

The concept of **direct** event correlation was used in SENACTIVE InTime [9] even before we started developing EventCloud. SENACTIVE InTime heavily relies on event correlation for complex event processing. An event service is allowed to define a correlationset on its incoming event types. If an event arrives, it activates the matching correlation session allowing the event service to retrieve or store date to/from the session. In comparison to our implementation, the scope of a correlation definition is defined for a single event services, whereas we define correlations globally for event types. A correlation session in SENACTIVE InTime

is always used in a single event service, whereas with EventServer the same session is used for all event services that process the event. Chapter 5.3 discusses this design difference in more detail.

In my opinion, ESP and stream languages are just a part of complex event processing. The stream engine (like Esper) is just another event service in our EventServer system like any other event service. Nevertheless, the beauty of the concept of declarative queries commands respect. Some parts of this concept were borrowed, as we will see in Chapter 7, where metrics are declaratively defined on top of correlated events.

4 Event Information Retrieval

In Chapter 2, we have seen that a relational schema is not appropriate to represent events for analysis. I pointed out that it would be better to have a data schema which directly represents events. This chapter will outline the concept of how events are organized in EventCloud in a document-oriented data representation with the help of event correlation. This approach allows querying for (correlated) events by keywords, building the base for event analysis in the EventCloud frontend.

This chapter covers the concepts behind two essential parts of EventCloud: *Event indexing* and *event search*. To make an event searchable it must be first added to the data representation and to the full-text index which is used to search for events. This step is called *event indexing*. *Event search* is executed against the full-text index of the data representation and is the main tool for a user of the EventCloud frontend to support event analysis.

Event indexing and searching is a concept to which not much attention was paid until now. The main topics the industry and research focus on are complex event processing (CEP) and event stream processing(ESP) which primarily deal with the processing of large event streams in real-time. However, publications on these topics disregard the large amount of event data that is generated with CEP. Input events are used by CEP for real-time monitoring and decision making. New events are generated by CEP showing which decisions have been made as a result of the input events. All events together show fine grained process data that is not available in a data warehouse at this level of detail. We are convinced that the event data is more than a transient during CEP, but is the basis for a new generation of

analysis tools.

Events offer a large amount of data for search, review and analysis of detailed information on the finest level of detail. Any relevant activity in a process is represented by an event reflecting the whole process, if correlation is applied to relate the process events. EventCloud is set on top of the low level event data and enriches the databasis through event correlation, metric calculation and aggregation.

When we started working on EventCloud, we drew a comparison to the internet search engines. The term "Search" is nowadays closely related to the big players in the world wide web in such a way that some time was needed to emancipate the concept of EventCloud from it's big brothers. Nevertheless, we stayed close to web search engines in some areas of EventCloud: a simple textbox with minimal options should be sufficient for complex event search. As will be seen in Chapter 8, we have adhered to this principle.

As time passed, and our knowledge and experience in the area of event search increased, we realized that in some parts we struggle with problems different than those of the web search engines. In the literature we found more and more interconnections to the classical areas of Information Retrieval. As Information Retrieval has evolved during the internet-hype of the last years, a new sub discipline "Web Information Retrieval" arose. Whereas classical IR provides the foundation, "Web Information Retrieval" extends the topic around web search engines. Event indexing and searching is a new concept that we need to classify within the area of information retrieval. To show the interconnections of EventCloud to Web IR, as well as classical IR, I like to compare what I call "Event Information Retrieval" to these two existing areas of research in the rest of the chapter.

Baeza-Yates and Ribeiro-Neto characterize Information Retrieval in [3] as

> *"Information Retrieval(IR) deals with the representation, storage, organization of and access to information items. The representation and organization of the information items should provide the user with easy*

access to the information in which he is interested.", Baeza-Yates, Ribeiro-Neto [3]

This is a pretty good description of what EventCloud has to deal with. The *information items* within EventCloud are *events*. Thus EventCloud has to care about the representation and organization of events in its system. Correlation is one of the key concepts we use for organizing events. The easy access for a user is a consequence of the intelligent organization of events in the EventCloud system. We will see more about this in the subsection 4.1.2.

A second definition accents the importance of *searching*. This definition is more affected by web information retrieval, so it points out the concept of searching documents that can be linked to each other which store meta data and are of different document types. Because events are connected to other events via correlations, EventCloud implements features highlighted by this definition:

> *Information retrieval (IR) is the science of searching for information in documents, searching for documents themselves, searching for meta data which describe documents, or searching within databases, whether relational stand-alone databases or hypertext networked databases such as the Internet or intranets, for text, sound, images or data.*[1]

4.1 Index & Retrieval Process in EventCloud

Baeza-Yates and Ribeiro-Neto define in [3] a index and retrieval process for IR systems that I want to compare with the process of event index and retrieval in EventCloud. We will see that EventCloud follows the same principles with slightly different characteristics.

4.1.1 Definition of the Input Data

Whereas classical information retrieval systems get their input from relational databases storing (text-) documents, todays web retrieval system have to follow

[1] http://en.wikipedia.org/wiki/Information_retrieval

another approach. Search engines have crawlers that scan the Internet to find interesting documents. Every now and then, these documents are used to build a new search index replacing the old one.

EventCloud neither retrieves documents from a data store (such as a relational database, a filesystem), nor does it crawl for information. Rather, we have to define the channels where events arrive to the EventCloud system. Events are generated by different IT systems and so must be gathered from different sources to add them to the EventCloud index. Therefore we have to define adapters where events are received from the systems where they were generated. EventCloud realizes this requirement by offering interfaces to existing middleware systems like message queues. Any event-generating system can connect to the message queue and send its events to EventCloud. If EventCloud is used in association with a CEP software, the CEP software would send the relevant event that should be available for analysis to the EventCloud adapters.

EventCloud uses typed events that have a well defined structure, such as the an event type defined in Figure 3.3, to select specific events that should get indexed. Event correlation sets these event types into something coherent.

4.1.2 Definition of the Documents - Mapping Events to Documents

Documents are the units of information typically represented in an IR system. If a document fulfills a search query, it is returned in the resultset. The information units EventCloud receives are events, so they are utilized as the units of information within EventCloud.

In comparison to other documents like books, manuals, emails or webpages, an event is a very small unit of information. Whereas a book tells a whole story, a single event only represents a short activity in time storing only data to this specific activity, and per se no relation to other events is available. So, the simplest approach is to build a new document for every event and add this document to the index. The mapping between documents and events is 1:1. However, this is

not sufficient for all retrieval tasks and will be discussed in more detail in Section 4.2

For example, the *TransportStart* event, as shown in Figure 3.3, represents a very significant activity but does not provide information about its relation to other events, like a matching *TransportEnd* event. However, most times a background-story can be told for an event, as it is often caused by other events and is causing further events. In a complex event processing scenario an event is connected to numerous other events. We have seen this when I described the meaning of the bridged correlation *AllOrderInformation* in Section 3.2.4. Correlated events, in comparison to single events, tell a story about a business transaction, a customer relationship, or some other business process instance. Event Correlation can be used to build documents of interest on the fly as it is most natural for a user to search for such business episodes. This is a fundamentally different approach to classical IR: whereas IR, as well as Web IR, process existing documents, EventCloud builds it's own documents. These documents are defined through (bridged) correlations and allow to coherence events to larger *"Documents of correlated events"*.

So besides documents that only store a single event, we also build correlation-documents to organize the information items (the events) in a more sufficient way. That follows the quote from [3] that the organization should support the easy access to the information items for a user. When users of EventCloud are interested in more than just single events, building larger documents of correlated events helps them to retrieve relevant data for complex event analysis. Section 4.2 discusses the structure of documents in EventCloud in more detail.

4.1.3 Creation of the Inverted Index

As the documents in EventCloud have been defined in the previous step, they now can be added to the inverted index. The representation of documents in the index is shown in Figure 6.3. Because index creation is a complex task, we rely on available tools specialized for this IR step. Apache Lucene is used to manage

the full-text index. Chapter 6 discuss the inverted index and the procedure to add events to it in more detail.

One of the main features of EventCloud is the real-time updates of the inverted index. Updates are event driven, i.e they are triggered on every new event so the event's information is immediately available in the index when the event is processed. This means the event, as well as all the documents it affects, is updated in the index in real-time. In comparison, most IR systems update their index in periodical cycles, or replace their index with a new revision, like web search engines do. It was a requirement to make EventCloud real-time to distinguish itself from other analytical tools which always have a gap between the real world and the data represented in their repositories.

4.1.4 Information Retrieval Model

Different models are known for information retrieval systems. In the literature for classical IR the *Boolean Model*, the *Vector Model* and the *Probabilistic Model* are defined. [17] compares and discusses these models.

EventCloud is realized with the *Boolean model* as it internally uses Apache Lucene. However, the presented concept of event indexing and searching could also be realized with another model. In opposite to the other two models, the boolean model works with the method of *exact match*. A document is in the resultset of a query if and only if it matches the query exactly. That is the case if all search terms are found in the document. In Section 4.2 the impacts of the boolean model on the search results are illustrated.

The boolean model typically allow the operations *AND*, *OR* and *NOT* to be used with the search terms. By using the Lucene query language[2], we have a larger set of operations available. Search terms are connected with an *AND* operation if not provided otherwise.

[2] http://lucene.apache.org/java/docs/queryparsersyntax.html

Per se the boolean model does not rank results, but only returns documents which match the search query in a random order. As this approach is not sufficient, we have to introduce ranking documents in the resultset. Ranking is a consequence of the large resultset a query in an IR system potentially returns. As most web search engines also use the boolean model for their search, they were forced by the sheer amount of data in the internet to implement complex ranking algorithms. According to [17], the average user search query in the Internet has a length of 2,6 terms. With this user behavior and the size of todays search indexes, resultsets without ranking would not make much sense. [17] discuss ranking algorithms for web search engines.

As the amount of data managed by the EventCloud system increases with every new event, EventCloud has to implement ranking, too. However, the ranking in EventCloud can only partially be compared to ranking algorithms of web search engines. Events are different to documents in the World Wide Web, but similar to the concept of hyperlinking, which is the most famous input parameter in Google's Pagerank described in [5], event correlation can be used: events are linked to other events via defined correlations. These events are again correlated to other events. This perspective on event correlation allows us to span a net of events starting from a single event. Figure 6.2 demonstrates how a single event is directly and indirectly linked to multiple others. EventCloud use this characteristic to enhance the possibilities of event analysis.

At the documents level, which can be a single event or multiple, correlated events, we rely on the ranking algorithm implemented in Lucene. For the exact formula see [11]. Additionally - on the conceptional layer of "documents of correlated events" - we have developed the search levels Rank1, Rank2 and Rank3. These search ranks are logical consequences of the *exact match* behavior of the boolean model and the characteristics of single and correlated events. See Section 4.2 for an in-depth discussion of this topic.

4.1.5 User Interface

The user interface is the point of contact for an ordinary user of EventCloud. As Google and the other popular search engines have created a de facto standard for search interfaces, we oriented on their user interfaces as shown in Figure 4.1.

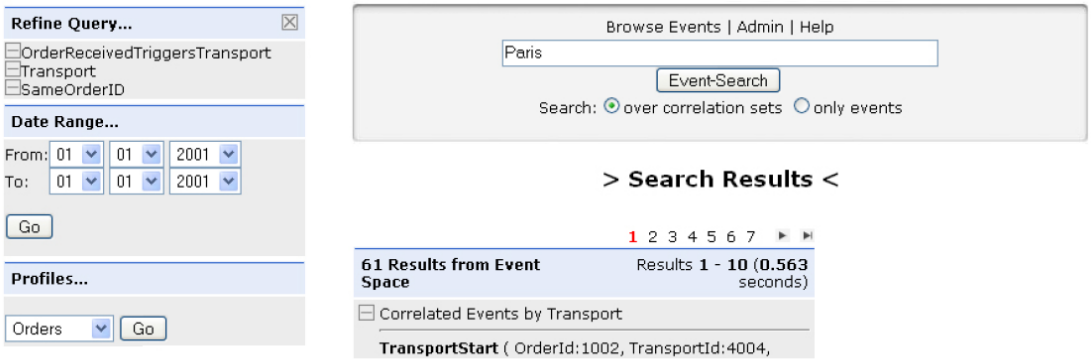

Figure 4.1: The EventCloud user interface

The search query is entered in a simple textbox. Additionally, EventCloud offers some user controls which allow setting filters on the searched event types and correlationsets and specifying a date range for the search query. These filters can just as well be coded into the search query, however that's not very handy for novice users. The resulting documents are displayed in a list ordered by their relevancy. To retrieve less relevant events, one has to page down to these results. Users are allowed to specify the search level(s) they want to execute their query against. As we will see in Section 4.2, each search rank can return different resultsets.

Results in EventCloud can be enriched by metric data calculated during event processing. This data is displayed along with the *detailed events view* in the *metric view*, as we will see in Figure 8.4. Because we realized that metrics on events becomes an increasingly important topic, we will might detach ourselves from the search-centric, Google-like user interface in future versions of EventCloud. It's our goal to move the *metric view* to the foreground and let event search only be the

basis for more analytical features. Because metrics are better understood with diagrams and charts, there is a need to plug in graphic controls and to provide high interactivity. With these requirements it is questionable if the next version of EventCloud will again be implemented as a web application. How future versions of EventCloud might look like is sketched in Section 5.4.

4.2 Resultset Ranking with Event Correlation: Rank1, Rank2 and Rank3

The previous section discussed how *documents of interest* are built for EventCloud on the fly with the help of event correlation. An example demonstrated why correlated events are needed to better understand the circumstances of a single *TransportStart* event. This section discusses the reasons why three search levels, *Rank1*, *Rank2* and *Rank3*, were introduced in the EventCloud system. The following section will then present an example which shows the behavior of the search levels.

EventCloud uses the method of *exact match* to retrieve the results for a given query. If every event is represented in a single document, only the events fulfilling the query are returned in the resultset. Furthermore, every event is isolated from all other events in the resultset. This may be accurate for some scenarios where a user wants to obtain a single event by its identifier, but in most cases this behavior would not be sufficient. A single event is such a small item of information that complex analysis cannot be done on a single event. Because each event is isolated, a user can not browse from a selected event to related events.

Until now we know, that events can be connected to each other via event correlation. Correlate events have a relationship with one another. The following kinds of correlation have been defined in Chapter 3:

Correlationsets All events in a correlationset are **directly** related. *The **TransportEnd** event is directly correlated to the **TransportStart** event with the same transportId.*

Bridged Correlation Events are **indirectly** correlated via multiple correlation-sets. They can not be related directly. *The **TransportStart** event is only indirectly correlated to the **Demand** event via the Correlationsets **shipmentInfo** and **demandToShipment***

EventCloud interprets these types of correlation to allow a ranking in search query results. If directly correlated events can fulfill a search query they have a higher relevancy than indirectly correlated events which can fulfill the same query. Directly correlated events are *strongly coupled*, so they fulfill a search query at a better level than *loosely coupled* indirectly correlated events.

With this definition the three search levels that exist in the EventCloud system can be determined. EventCloud internally manages a search index for each of these levels. Every index holds different documents as it represents different types of correlations. The technical description for managing rank indexes can be found in Chapter 6. Each level has its its specific purpose and can be useful depending on the concrete usage scenario:

Rank1 No correlation is used at all for Rank1. Rank1 is the full-text index over the processed events where each document in the index represents exactly one event.

With the method of *exact match*, a document is returned if all search terms are found in this document. Because one document in the index is representing exactly one event, a hit would be the **perfect match**. An event storing all the search terms a user enters is a direct hit. All search terms are concentrated in one event which means maximum coupling of the search terms. So a hit in Rank1 rates higher than any hit in Rank2 or Rank3.

It is rather unlikely that a document in Rank1 fulfills a complex query because a single event holds minimal information. The disadvantage of Rank1 is its low *Recall* value. There might be many relevant events for a search query if event correlation is taken into account, but because only single events are considered it

is likely that no result will be returned.

Rank2 Event correlation with correlationsets is used for Rank2. Rank2 is the full-text index over all *directly* correlated events. Each document in the index represents a set of events that are correlated via a correlationset, i.e. each instance of a correlationset is represented as a document in the index.

Fewer events are in this index than in Rank1 because only correlated events are added to this index. If an event type is not correlated in any correlationset, it is not available in the Rank2 index. Since an event can participate on multiple correlations, the event can be stored in multiple documents representing different correlations.

A document is returned if all search terms are found in one document, meaning that the search terms must be found over directly correlated events. A hit in Rank2 is rated lower than any hit in Rank1 because the search terms are only found across multiple events, but rated higher than a hit in Rank3 because search terms are found in directly correlated events.

Rank3 Bridged correlation is used to create Rank3. Rank3 is the full-text index over *indirectly* correlated events. One document in the index represents a set of events that are indirectly correlated via a bridged correlation across multiple correlationsets, i.e. each instance of a bridged correlation is represented as a document in the index.

Fewer events are in this index than in Rank1 because only indirectly correlated events are added to this index. If an event type is not correlated in any bridged correlation, it is not available in the Rank3 index. However, an event can participate on multiple bridged correlations, therefore the event can be stored in multiple documents representing different bridged correlations.

A result is returned if all search terms are found in one document. That means

that the search terms must be found in events that are at least correlated indirectly. Search terms must not be found in events that are directly correlated. It's sufficient that the terms are found in events that are related over a *bridging event*. A hit in Rank3 is rated lowest, although Rank3 may returns results where Rank1 and Rank2 are not able to return any hit. This makes Rank3 important for EventCloud.

4.3 Example for Event Search on Rank1, Rank2 and Rank3

Using the correlation *AllOrderInformation* from Figure 3.2, we explain the meaning of the search levels when executing search queries against the EventCloud index. The following queries should be executed to show the characteristic of each rank:

- "Madrid"
- "Madrid and Vienna"
- "ProductA and MegaTrans and Acceptor=Jane Smith"

Figure 4.2 shows the documents that are available in each search level index. For this example the four events have been indexed the following way:

- Every event is indexed individually for Rank1 index.
- The correlationsets *demandToShipment*, *shipmentInfo* and *transportInfo* are used for Rank2. This results into three documents, one instance of each correlationset.
- The bridged correlation *AllOrderInformation* is used for Rank3. The four events can be correlated indirectly, so they are represented in a single document.

The different Ranks represent the same events in different ways according to their interpretation of event correlation. As in Rank1, the mapping "event to document" is always 1:1, where in Rank2 and Rank3 a single document can hold multiple events. Because all search terms must be found in a resultset's document it follows

that the searchranks must return different resultsets. Rank2 and Rank3 store less documents than Rank1, thus the documents are bigger and may grow if a new event arrives. In Rank2, a single event is stored in multiple documents[3] (e.g. *ShipmentCreated*, *TransportStart*). This redundancy is needed and cannot be avoided in this approach.

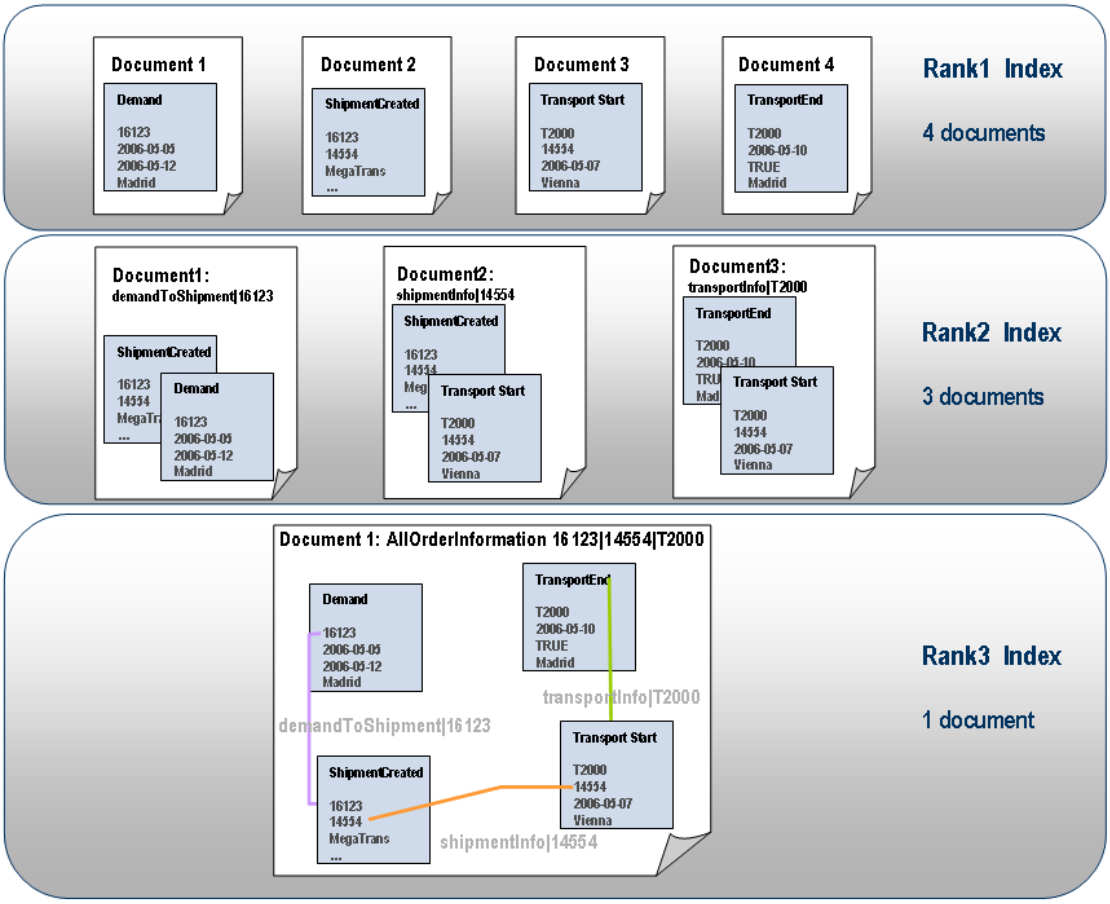

Figure 4.2: The documents in the three search ranks

[3]this can also happen in Rank3 if an event participates on multiple bridged correlations that are not connected to each other

4.3.1 Rank1 - Without Correlation

Rank1 is a simple full-text index over events. This search category is similar to the search over text documents. If all search terms occur in an event, it is returned as result. Unfortunately an event, as compared to e.g. a typical .html file, is very short with little data. This makes it less useful for event search.

Nevertheless, Rank1 is useful for a user of EventCloud searching for a specific event. If the user knows the right search term, it's a very efficient way to retrieve the event in which the user is interested. For a user who wants to search, analyze and navigate through events, Rank1 is not that useful. As no correlation is applied, an event is not related to any other event. However in EventCloud Rank1, searches can be used to retrieve a specific event which is then used to browse to other, correlated events.

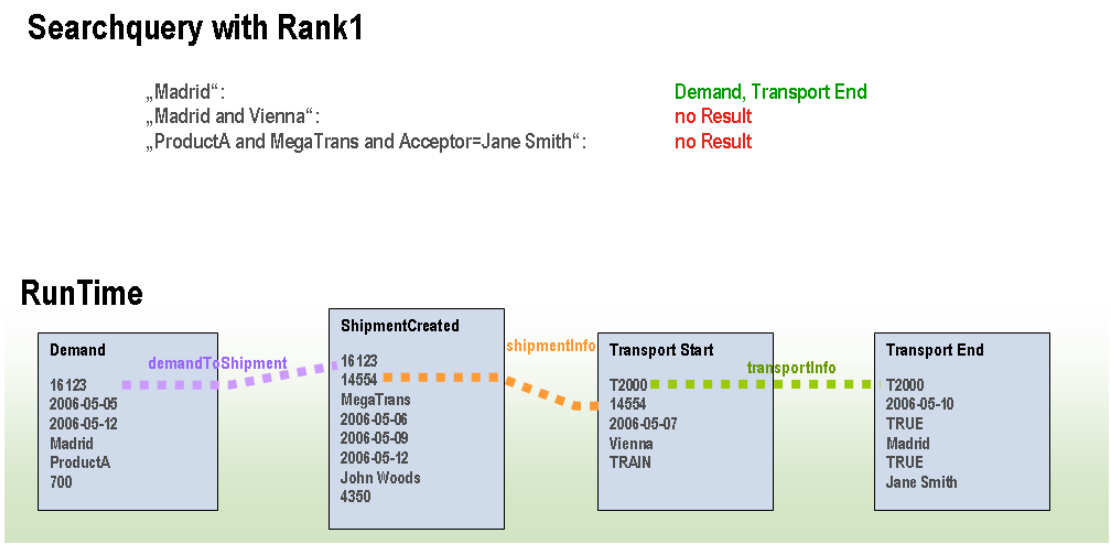

Figure 4.3: Returned results in Rank1 (Correlations are not used)

In Figure 4.3 Rank1 returns the two events *Demand* and *TransportEnd* which include *"Madrid"* in the resultset. For the two other queries, Rank1 is not able to return any result. Consider that in a real scenario numerous events with the

attribute value *"Madrid"* can exist in the index. Rank1 search is best suited if an event should be retrieved that is identified by a (nearly) unique identifier.

Rank2 will return the two documents *Document 1: demandToShipment|16123* and *Document 3: transportInfo|T2000* because *Madrid* can be found in these correlations[4]. Rank3 will return its document because *Madrid* is found, too. Yet these results have lower ranks than the results in Rank1.

4.3.2 Rank2 - Directly Correlated Events

Searching over correlationsets gives the user the possibility to search for multiple terms that occur in directly correlated events. The resultset includes all matching documents, where each document represents its correlated events.

Figure 4.4 shows that Rank2 will return a result for the query *"Madrid and Vienna"*. Rank1 does not return a result because no event has both search terms. However, the events *TransportStart* and *TransportEnd* which are correlated via the the correlationset *transportInfo* with the correlation value *transportid=T2000*

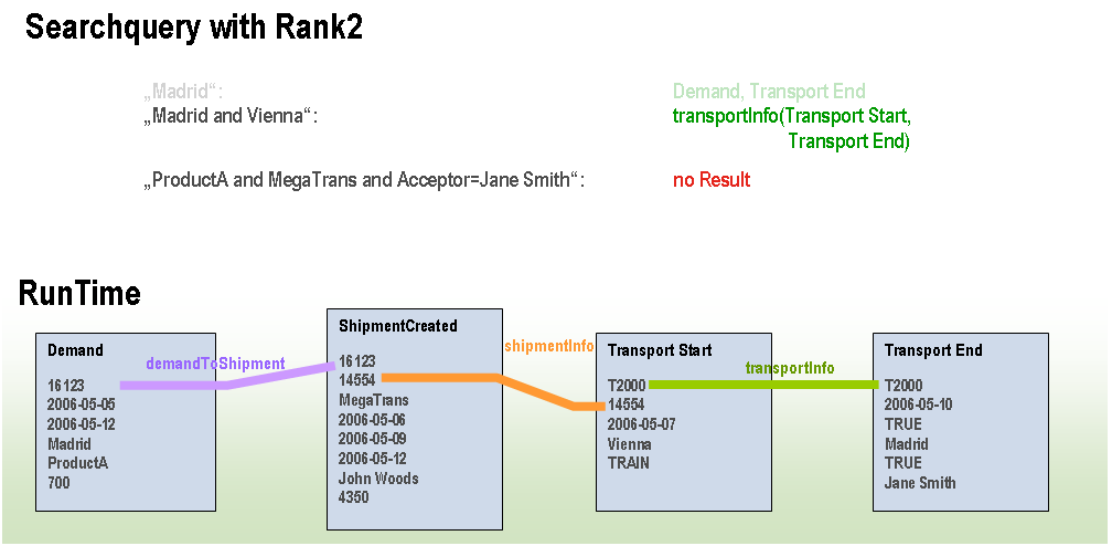

Figure 4.4: Returned results in Rank2 (classic correlationsets used)

[4]because the *Demand* and *TransportEnd* participates on these correlations

fulfill the query together. *Document 3: transportInfo|T2000* from Figure 4.2 will be returned for the Rank2 query. Also the Rank3 document is returned because the terms *Madrid* and *Vienna* can be found in it. Yet the result has a lower rank than the results from the Rank2 index.

From the user's perspective, Rank2 may returns *False Positives*. As the search terms are only found across correlated events, Rank2 may return documents other than those the user intended. This is a problem with inexact search queries and the amount of documents available in event search. EventCloud provides filters to avoid this problem.

4.3.3 Rank3 - Indirectly Correlated Events

Bridged correlations are a new feature added in this version of EventCloud, hence Rank3 is a new search concept the previous version of EventCloud does not include. It is defined as the lowest search rank, but is able to return results where Rank1 and Rank2 cannot recognize the coherencies. Rank3 allows a user to search for multiple terms that occur in indirectly correlated events. The resultset includes all matching documents where each document represents its indirectly correlated events.

With a Rank3 result, the search terms are loosely coupled across the events as events in a Rank3 document are only correlated via "bridging events". The chance that Rank3 results are *False Positives* is even higher than with Rank2. However Figure 4.5 shows that the *Document 1*, representing the bridged correlation *"AllOrderInformation"* from Figure 4.2, is returned by the query *"ProductA and MegaTrans and Acceptor=Jane Smith"*. The search terms are spread across indirectly correlated events, where the event *TransportStart* does not hold any search term, but acts as the bridging event to *TransportEnd*.

Because Rank1 and Rank2 cannot fulfill the search query with one of their documents, the Rank3 result is the only, and highest rated, result in the resultset.

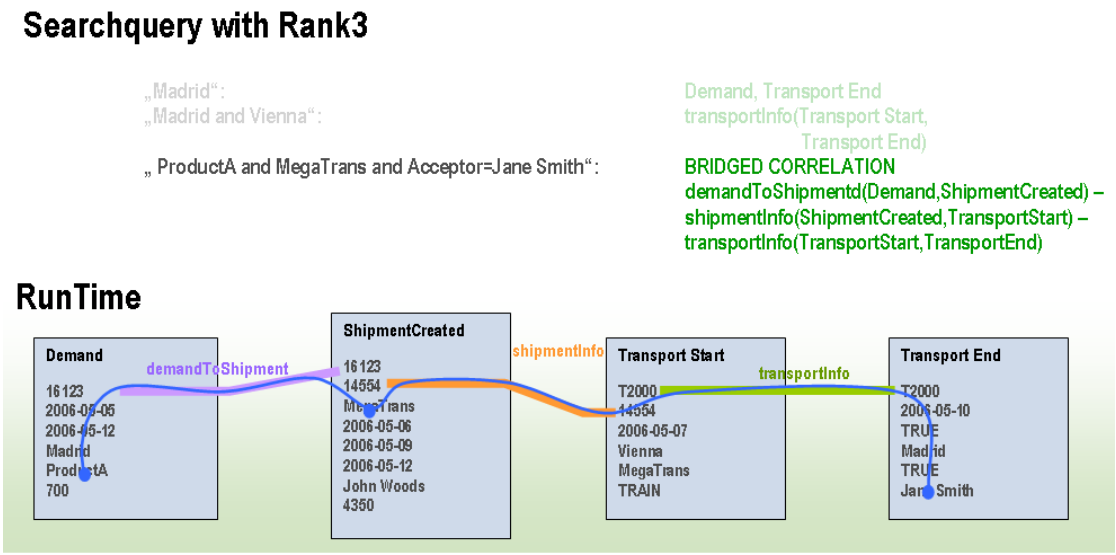

Figure 4.5: Returned results in Rank3 (bridged correlationsets used). The figure shows how the search terms are found across the events in the correlation.

4.3.4 Conclusion

In EventCloud, the three ranks are used as conceptional ranking of query results. Documents returned by Rank1 are always rated higher than documents returned by Rank2. Documents returned by Rank2 are always rated higher than documents returned by Rank3. The resultset in EventCloud shows first the results of Rank1 search, then the Rank2 results and lastly the Rank3 results. Within each of the three resultsets, we rely on the Lucene ranking algorithm.

Sometimes users want to execute their query at a specific search level, so the possibility exists to search in a specific rank. Rank1 can be less than optimal if a single, generic search term is used. The goal is to transparently use the search levels to return the best results for a user. Advanced users can use the levels to immediately retrieve the documents which are relevant to them.

4.4 Characteristics of Event Information Retrieval

[17] compares Web Information Retrieval with classical Information Retrieval. Following this comparison, a column, *Event Information Retrieval*, is added for the approaches described in this chapter and implemented in EventCloud. Table 4.1 shows how *Event IR* behaves in relation to the these well defined areas of IR.

EventCloud has well defined documents just as classical IR has; no crawling through an unknown space is needed to obtain relevant documents. The input events are pushed to the EventCloud system to be indexed. The EventCloud index is always complete for all processed events and is updated in real-time by every event.

In classical IR, every document stands on its own. With web retrieval, the link structure of the web is heavily used to make a relevance ranking. With event search, the documents are also linked to each other because an event can correlate to other events in different event correlations. From a single event, a user can browse to other correlated events (linked documents).

Ranking is needed for Event IR because the amount of events and the simple query language, potentially cause large resultsets. EventCloud has conceptional search levels presented in Section 4.2 and a ranking within these levels calculated by Lucene. Search operations and user interface are purposely built like popular search engines to allow novice use. EventCloud is primarily relevant for analysts and experts that need detailed information about a business scenario, so the user interface may change in future development steps.

	Web IR	**Event IR**	**Classical IR**
Length and granularity of the documents	Length of the documents varies, large documents are often split	Length of the events is very short, with concentrated information. As events are correlated, a single document representing this correlation can grow large. An event can be represented in multiple documents(correlations) in the system.	Length of the documents varies within a range; each document is represented by a single document in the IR system
Hyperlink structure	Documents are linked to each other	Events are linked to each other via correlation. Correlation does not automatically increase the relevance of an event, but allows more complex queries.	Normally documents are not linked. Their is no necessity to infer the quality of an document from it's link structure
Size of the dataset	Overall data size unknown. Completely indexing is impossible	Database is growing with every event. Index is complete for all event types and correlations that are defined to be indexed.	Index is complete for documents that should be indexed.
User query types	Numerous user groups with different needs	Well defined user groups with well defined needs.	Well defined user groups with well defined needs.
User Interface	Simple user interface, understood by novice users	By now simple interface, understood by novice user. But may get more complex if more analytic features are implemented.	Complex interface; tutorials and practice needed
Ranking	Relevance ranking because of the sheer number of results	Relevance ranking needed because of the sheer number of events generated by a scenario. Additional the concept of Rank1, Rank2 and Rank3 exist to differ between the levels of correlation.	Relevance Ranking not always necessary. Well-defined search queries causes smaller resultsets.
Search operations	Simple search operations, used by most users.	Simple search operations. Some standard filters to allow user to narrow their query on event types, correlationsets.	complex search languages

Table 4.1: Comparison Web, Event Search, classical IR

5 Building an Infrastructure for Event Analysis

5.1 Evolution of the EventCloud System

EventCloud was first introduced in the master thesis of Szabolcs Rozsnyai [28]. Starting in summer 2005, with the development of scenario-examples for EventCloud (see appendix of [28])and going through multiple phases of design and redesign, Rozsnyai developed a first proof-of-concept prototype of EventCloud.

EventCloud included a web interface giving a user the power to search terms in events (Rank1), as well as in correlated events (Rank2). The backend server implementation went through multiple architectures giving a good background on the topic. Many points from this first implementation were learned which are discussed in this section, along with conclusions and a presentation of the new implementation.

5.1.1 EventCloud's Original ETL Architecture

In a nutshell the original version of EventCloud extracts already correlated events from a database, splits the events' attributes, indexes this data to a reverse index with Apache Lucene and stores the events in the EventCloud database.

Figure 5.1 shows the high level overview of this architecture:

IntimeDb Source Events This database stores all the events EventCloud is able to index. Therefore, all events plus their correlation information are stored

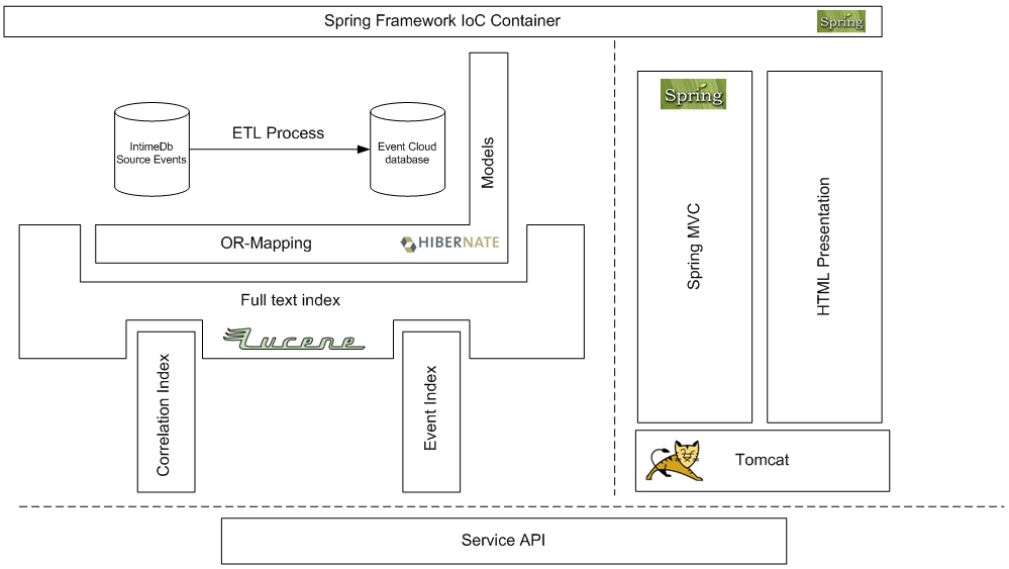

Figure 5.1: Old architecture of EventCloud

in this database. EventCloud uses this already available knowledge to build a valid Rank1 and Rank2 full-text index for these events.

ETL Process The ETL (Extract Transform Load) process is a user triggered process to start extracting events from IntimeDb, transforming them (which basically means splitting event attributes) and loading them into the reverse full-text index and the EventCloud database. OR-Mapping and the creation of the full-text index are executed during the ETL-process

HTML Presentation On top of the Service API, a web user interface was created. The user interface provides the possibility to manage EventCloud (starting the ETL-process, changing the index location, etc.) and it gives a user the facility to search through the indexed events.

For a more detailed description, refer to [28], which gives a broader overview on the architecture as well as the on the technologies used.

5.1.2 Critics on Architecture

Figure 5.1 shows how the ETL-process takes a central position in the whole picture. Triggering the ETL-process moves the events from the InTimeDb into the EventCloud system. With the benefit of hindsight, this approach was questioned, as it is very monolithic and static, if it is sufficient to the requirements of EventCloud.

- The ETL-process depends on the correlation knowledge stored in the InTimeDb. The concept of event correlation must be applied before the events come into the EventCloud system.
- The ETL-process is a heavy weight, inflexible, not easily extendible process. It executes the following tasks in one big step:
  1. Select all events + correlation information from database.
  2. Splitting the events attributes.
  3. Storing event attributes in EventCloud database.
  4. Storing events in Rank1 index.
  5. Storing events in Rank2 index.
- The ETL-process is an old fashioned batch-insert job, comparable with a batch load into a data warehouse. It does not follow an event-driven approach.

With these problems and restrictions in mind, we came up with the following demands which were to be considered for the new version of EventCloud:

- Implementing an own event correlation service for EventCloud, including new ideas like bridged correlation (Rank3).
- Do not batch insert data; the system needs to work in real-time. A received event should be processed immediately. EventCloud should become event-driven, because this is the most natural way for an event processing system.
- Lightweight "event services" instead of a single heavy weight ETL-process.
- Increase the performance in comparison to the original ETL-process.

System Service: Event Correlation

As mentioned previously the ETL-process depends on the event information in the InTimeDb database. InTimeDb was created and filled by the prototype of the commercial CEP software SENACTIVE InTime:

SENACTIVE InTime includes a system service that implements the idea of correlation (without bridged correlations) as described in Chapter 3. This concept is the basis for EventCloud's Rank2 search. When Szabolcs Rozsnyai started developing EventCloud, he used the SENACTIVE InTime prototype not only for simulating events, but also to correlate the events that should be indexed in the EventCloud system. The SENACTIVE InTime prototype processes these events, generates the event correlation and persists the events and their correlation information to the *InTimeDb*. Afterwards EventCloud picks the data from the database to generate the full-text indexes.

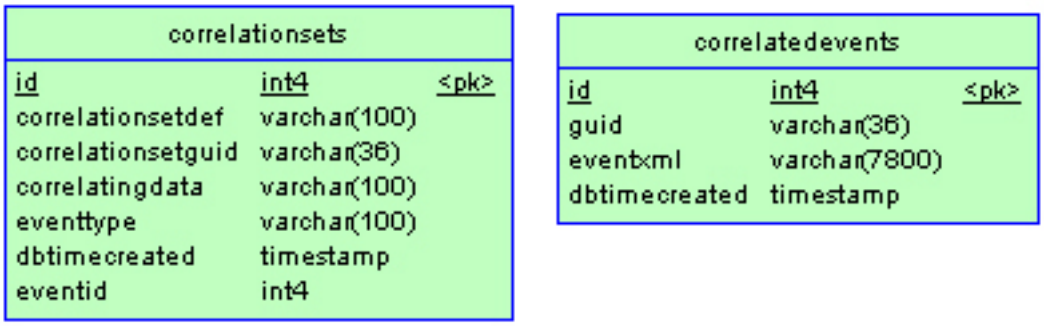

Figure 5.2: InTimeDB stores all events and their correlation. EventCloud extract information to generate full-text indexes[28]

Every event is stored as a XML representation in table *correlatedevents*. For every correlation session in which an event participates, an entry in table *correlationsets* exists. EventCloud takes this information and has an easy task to build the matching Rank1 and Rank2 full-text index. However, this approach also has major drawbacks:

- SENACTIVE InTime prototype is just what it's called: a prototype. No further development will be done for this piece of software. It's a closed-source, proof-of-concept for the commercial SENACTIVE InTime product line. EventCloud should not depend on this prototype for such essential parts as event correlation and Rank2 search (and furthermore Rank3).
- Every event added to the EventCloud system needs to be processed by the SENACTIVE InTime prototype first. Using two systems for making one task, namely indexing events for a search index, signalizes something is wrong. Using two systems makes everything more complicated: two installations, two places to define events and two places to define correlationsets.

We decided to move system borders: EventCloud should be a system that has all needed parts on it's own. The essential correlation service has to be implemented for EventCloud, including many extensions to realize the new ideas of bridged correlations. InTimeDb should no longer be required for newer versions of EventCloud. Instead, events should be directly received and executed by EventCloud in an event-driven way, applying correlation immediately when an event is processed.

This approach gives us a clear cut between SENACTIVE InTime and EventCloud. Nevertheless, SENACTIVE InTime remains a significant tool for research with EventCloud: the SENACTIVE InTime Simulation Studio allows to generate events and publish them as event streams. The Simulator is a mighty tool not only for testing but also for creating (correlated) events for complex event stream scenarios. The simulator was, for example, used to generate the events in the Medicare Example discussed in Chapter 8.

Batch Processing of Events

This section is related to the previous one. As system borders were moved, there was the chance to completely replace the ETL approach. The original architecture did not know when a new event arrived. The implementation was absolutely not event driven. The SENACTIVE InTime prototype stored events in the InTimeDB while EventCloud was waiting. Then the ETL-process was triggered by

a scheduled job or was forced by the user. One chance would have been to poll the database if a new event was received to trigger the ETL-process, but that would not have been very elegant too.

The new EventCloud architecture is completely event-driven. If a new event arrives, it starts its own processing logic as fast as the needed system resources are available. An event is added to the search index as soon as possible (at least in theory: the problems which arise with this approach will be discussed in Section 6.3.2).

Whereas some can say *"it's not that important that an event is available in the searchable index in real-time"*, it's an important requirement defined for EventCloud. It's one of the main points where EventCloud differs from many products available today. Moving a step forward from the well-known data warehouse batch-update approach is a crucial point. Making data available in analysis software like EventCloud, BPM (Business Performance Management [10]) or data warehouses in near real-time is an essential requirement for modern decision support systems. Big players, like Microsoft [1], set a course for real-time business intelligence. The next chapters point out the essential need for an event-driven approach for further features of EventCloud.

User Defined Event Services

During development of the first version of EventCloud, numerous ideas for EventCloud were developed and it became clear that the current monolithic block "ETL-process" did not fulfill our requirements.

In the original EventCloud architecture, it was never planned to expand the "Event Search" capabilities with additional, different features. Therefore the "ETL-process" was created as a heavy-weight process doing all the indexing and persistence work. However, as time passed, new features for EventCloud emerged: metrics should be calculated for correlated events to allow a higher level view on fine-grained events, research for rule-based services on top of (correlated) events

[1] http://www.microsoft.com/technet/prodtechnol/sql/2005/rtbissas.mspx

should be done, and the concepts of data mining may be applied on events. EventCloud should be an open architecture allowing to easily implement user defined functionality.

To fulfill these wishes, the concept of event services is introduced. An event service takes one event in its *.process()*-method and executes the implemented task. Multiple event services can be executed for an event one after another, so every event service only needs to implement a defined part of the whole processing logic for an event.

We realized that the main backend topics of EventCloud, indexing for Rank1, Rank2 and Rank3, are just three event services we have to implement. Adding more event services, such as services that calculate metrics, makes EventCloud a rich platform for many topics related to event processing. To make a clear distinction between the *event processing platform* and the *event analysis application* created on top of this platform, the platform is called **EventServer**. Rank1, Rank2 and Rank3 event services running in EventServer, plus the analysis frontend, are the **EventCloud** system. So EventCloud is just one application that can run in the EventServer infrastructure.

A detailed description of EventServer can be found in Section 5.3. More on event services can be read in Section 5.3.5.

5.1.3 Performance Issues

As the original EventCloud was developed as a proof-of-concept, high performance was not one of the main goals. As EventCloud should work with a high number of events in complex scenarios, some tests were performed with the old architecture.

Starting the ETL-Process to index 1000 Events for Rank1 needed up to 20 minutes at 100 percent CPU utilization. That was unacceptable as this would mean less than 1 Event per second, whereas our goal was a two-digit number of events per second for creating all of the indexes: Rank1, Rank2 and Rank3. As the perfor-

mance numbers on Lucene's webpage[2] suggested a far higher performance, there was a question about the numbers measured with EventCloud. After doing some source code review and profiling with JProfiler, the reasons were quickly found:

- Extracting the attributes of every event and inserting them in the EventCloud database cost a lot. Say an event has 10 attributes in average, this would cause 10 insert statements for every event. As described later, the same functionality can be implemented for EventCloud without extracting and storing all attributes as an own entry in the database[3].
- The usage of the Apache Lucene API was weak at multiple points: Even though the ETL-process does a batch job it misuses the Lucene API. Instead of adding all events to the Lucene index with a single bulk write the following code was executed:

For every event in the ETL-Run:

1. Create new *IndexWriter* for the Lucene index.
2. Add a single event to the index.
3. Call *IndexWriter.Optimize()* on the index.
4. Close *IndexWriter*.

For every event the Lucene infrastructure is initialized new and *IndexWriter.Optimize()* is called each time. However, that does not make any sense when working with Lucene. According to [11]:

> *"It's important to emphasize that optimizing an index only affects the speed of searches against that index, and does not affect the speed of indexing."*

It's getting clear why *IndexWriter.Optimize()* is a performance bottleneck:

[2] http://Lucene.apache.org/java/docs/benchmarks.html
[3] see [28] 4.1 Database and Schemas

> *"[..]while optimizing an index, Lucene merges existing segments by creating a brand-new segment whose content in the end represents the content of all old segments combined [..] Consequently, just before the old segments are removed, the disk space usage of an index doubles because both the combined new unified segment and all the old segments are present in the index."* [11]

So be aware that calling *IndexWriter.Optimize()* during a batch insert to the Lucene index makes no sense. Optimizing the index should happen, if at all, after the batch insert, because in EventCloud's ETL-approach it is guaranteed that the index will not change for a while (until the next ETL-process starts).

To call *IndexWriter.Optimize()* in an event-driven system, like the new EventCloud approach, is much harder to decide: it's unpredictable when the next event arrives that will update the index. So there is no optimal point in time to optimize an index. This problem is further discussed in Section 6.3.

Opening and closing Lucene's IndexWriter for every event is useless too. It obtains and releases a file-lock on the index for every event, even though it's guaranteed that only EventCloud requests an IndexWriter in the current configuration. The right approach must be to create an IndexWriter once in your application and hold it open as long as possible (i.e. for the duration of the batch job), otherwise Lucene will spend more time in index locking than in indexing events.

For the new version of EventCloud, it was decided to use the Compass API[4] that encapsulates the complexity of the Lucene API to avoid misuse with the utmost probability.

5.2 Goals for New Infrastructure

The previous section discussed the lessons learned from the first implementation of the event search back-end. The architecture and our experience working with

[4] http://www.opensymphony.com/compass/

Figure 5.3: ETL-Process needs 95,9%

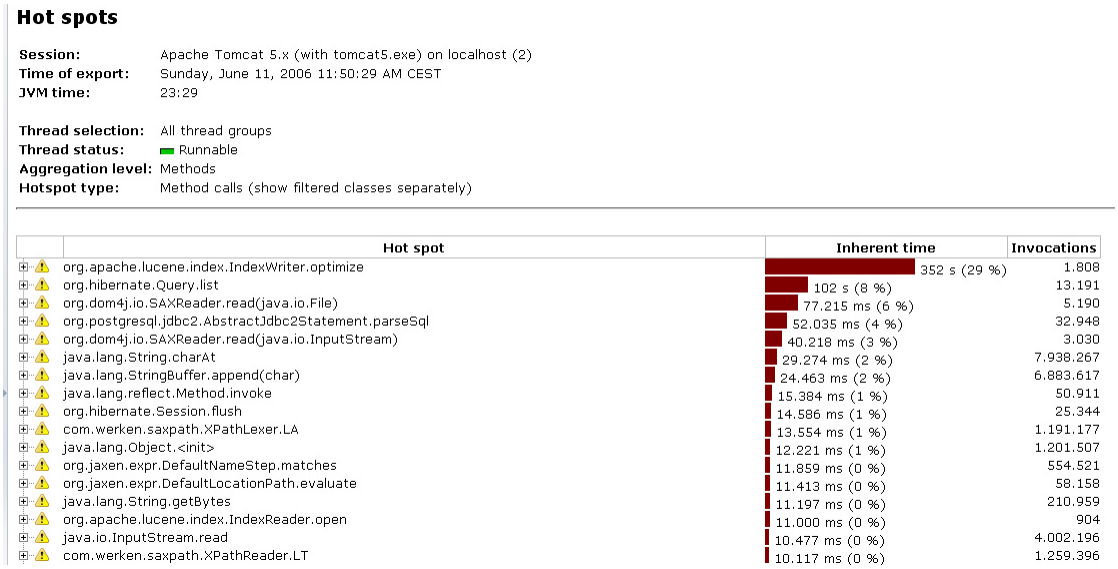

Figure 5.4: Lucene's *optimize()* and database persistence as hotspots

the topic were limited. Since then, we gained insight on the topic allowing us to better define our requirements.

Following is a summary of the main goals and benefits of the upcoming implementation:

Implement Correlation as reusable Service Event correlation, as described in Chapter 3, is the foundation for EventCloud and other topics on event processing. For the new EventServer, an independent, easy to use correlation service should be implemented. EventCloud suffered from its dependency on the closed source SENACTIVE InTime product. Implementing an own correlation service decouples our tool from any other systems.

Rank3 Implementation Hand in hand with the development of the correlation service, the idea of indirectly correlated events through "bridged correlations" should be realized. While a correlation is defined between a number of event types, a bridged correlation consists of multiple correlations to allow relating events over multiple correlationsets (Section 3.3). Indirect event correlation leads to an additional search level Rank3 described in section 4.2.

The users of EventCloud should get new power for their event search. While this idea already existed during development of the first EventCloud, we initially discarded this feature. Implementing bridged correlation adds new complexity to the correlation service. Obstacles that need to be considered were described in Section 3.3.

Metrics for Event Search and Analysis We found out during the first EventCloud development that event search may not be sufficient in many analysis scenarios. The results of an event search are often too fine-grained to satisfy the user requirements even when correlation is applied. As discussed in Chapter 2, we want to enrich the event search and analysis capabilities with the calculation of metrics.

More details about metrics, different approaches to implement metrics and

their implementation in the EventServer can be found in Chapter 7.

New Architecture: EventServer Because of EventCloud's interconnection with SENACTIVE InTime and the need for further research required a completely new architecture for EventCloud. The decision was made to implement a more generic platform where event indexing and search are just services as many other's might be. The following parts were identified:

EventServer the EventServer is a multi-threaded server that receives events at its adapter, provides system services (like event correlation) and executes configurable event services in an event-driven way.

System service System services are services that should be usable from everywhere in the EventServer. For example, the correlation service is implemented as a system service so every other service in EventServer can use event correlation.

Event service An event service takes one event, optionally retrieves its correlations and executes the defined *process*-method. The EventServer knows for every event received which event service to execute. So Rank1/2/3 index creation for EventCloud will be implemented as event service, while calculating a *Duration* metric is just another event service.

Following the approach of event services, the EventServer is a generic platform which implements any service that might be interesting for somebody in the area of event processing.

EventCloud Frontend As EventCloud is an analytical tool for users a frontend needs to be implemented. EventCloud's frontend was simple, clear and well understood by its users. We want to continue this direction, but have to extend the GUI on some points to realize the new concepts like Rank3 and metric data.

New Research With this version of EventCloud, a platform permitting further research on event processing should be created. EventServer provides a platform allowing everyone to realize their ideas on event processing. EventServer offers an easy way to deploy specific scenarios and applications that need an event processing infrastructure.

5.3 EventServer Architecture

The EventServer is the basis for implementing event indexing for EventCloud as well as other event processing tasks. The next sections will give an introduction to the new architecture.

5.3.1 Overview

The EventServer is an IoC[5]-Container for event processing. Events are received through an Event Adapter, processed in event services to which the EventServer offers system services. It is a highly configurable server allowing the exchange or addition of any system-/event service by implementing interfaces.

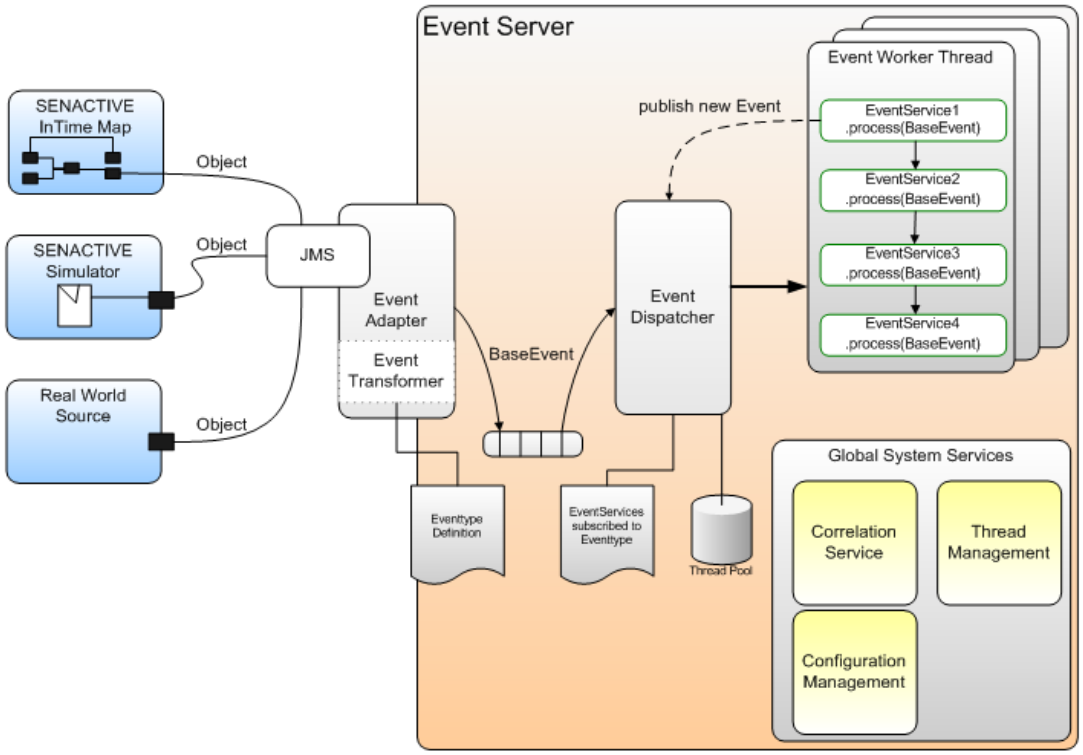

Figure 5.5: EventServer architecture overview

[5]Inversion of Control

Brief Event Processing Discussion

Events are received on the Event Adapter from different sources. Because events are normally generated in external IT-systems and received through different channels like queues, databases or webservices, the events are wrapped in diverse message types. Therefore, an event needs to be transformed, according to its event type definition, by the Event Transformer to map the message into the BaseEvent object used inside the EventServer.

The transformed event is added to an internal queue observed by the EventDispatcher. The EventDispatcher has a thread pool of EventWorker threads. If an EventWorker thread is free, the EventDispatcher picks the event from the queue and hands it to the worker. According to the event type, the dispatcher tells the worker which event services have to be executed with this event.

The EventWorker takes the event and the list of event services and calls the event services one after another. Event services implement some logic for processing the event.

Some of the event services make use of the EventServer's system services when they process the event. The system services can be looked up in the EventServer container.

When the EventWorker finished the last event service for a given event it is free for the next event. If a new event is in the queue the EventDispatcher will assign it to the worker.

5.3.2 Event Adapter and Event Transformer

The Event Adapter (including the Event Transformer) is the entry point to the server for every event. Because events occur somewhere in the real world they have to be forwarded to the EventServer.

In a typical IT-landscape, events can occur in many places:

- Every state change in a business process is an event.
- Every message in a network.
- Every new email in a mailbox could represent an event.
- A robot parsing webpages could generate events.

Events can occur in many different IT-Systems, at many levels of granularity, with different frequencies depending on the scenario one is interested in. As a least common denominator, we decided to implement a JMS adapter for EventServer. JMS is a standardized queuing system in the Java world, which allows us to be open for many different systems to receive their events.

However, the EventServer is not tied to the JMS Adapter. In the future, different adapters could be implemented easily. Currently the SENACTIVE InTime Simulator is used to simulate many of our scenarios. As the Simulator can send its generated events to a JMS queue ,the JMS Adapter is sufficient for now.

The Event Transformer receives a message representing the event from the EventAdapter. In our case it is a JMS message. This message is transformed to the internally used object BaseEvent which represents events in the EventServer System. Using SENACTIVE InTime Simulator, the events are represented by XML strings. To transform the events back to the Java object used inside of the EventServer, the transformer has to parse the XML string.

When the transformation is finished, the BaseEvent is put into an internal queue where it is picked up when the EventDispatcher has free resources to process the event.

5.3.3 Event Dispatcher

The EventDispatcher distributes an event to an EventWorker.

Listing 5.1: Event service subscription for the event type *TransportStart*

```
1  <EventType typeUri="eventobjecttype://Examples/Logistics-SpecialGoods/TransportStart">
2    <EventServicesSubscription name="IndexingSubscriptions3">
3        <EventService>EventService_Event2DBPersistence</EventService>
4        <EventService>EventService_ExecutionTimeMetricCalculation_transportInfo</EventService>
5        <EventService>EventService_EventRank1Indexing</EventService>
6        <EventService>EventService_EventRank2Indexing</EventService>
7        <EventService>EventService_EventRank3Indexing</EventService>
8        <EventService>EventService_LogPerformanceService</EventService>
9    </EventServicesSubscription>
10 </EventType>
```

At startup, a configuration file is loaded defining which event service needs to be executed for a given event type. Listing 5.1 shows the event service subscription for the *TransportStart* event type. Within the *<EventService>*-tags, the service names as defined in the EventServer IOC container definition are used, so the EventWorker can retrieve the event services from the EventServer container.

The dispatcher listens to the internal event queue. If an event is available, it dequeues the event and assigns it to an EventWorker out of its thread pool. The EventDispatcher looks up the event services which need to be executed according to the event's event type and hands the event service names to the worker. Then the dispatcher again listens to the internal queue.

5.3.4 Event Worker

EventWorkers are very simple. Every EventWorker is executed in a separate thread. A worker takes a single event and a list of event services from the event service subscription. Then, the worker calls the *.process*-method of every event service in the subscription. After finishing the last *.process*-method, the worker waits until the EventDispatcher assigns the next event (which may has another list of event services attached) to it.

5.3.5 Event Service

By now the infrastructure to process events has been described. Event services implement the logic one wants to execute in their event processing task. So for EventCloud services were implemented which can persist and index events. Other scenarios will require other event services. As event services are pluggable in

EventServer, other scenarios can be created easily.

Event services **process a single event** and are event-driven. An event service can be seen like a Servlet. An event service is always executed on a request, i.e. its *process*-method is called if an event should be processed. Like a Servlet, an event service can obtain sessions to the event's correlations to "load" a state. Event services make use of the concept of event correlation as it retrieves states set by previous events which correlate with the event currently processed. However, correlation is not handled by the event service itself; an event service has to call the system service *correlation service* which returns the correlation session for a given event and a given correlationset defined in the EventServer.

Listing 5.2: The event service interface to implement

```
1 public interface IEventService {
2   public void prepare(BaseEvent eventToProcess);
3   public void process(BaseEvent eventToProcess);
4   public void cleanup(BaseEvent eventToProcess);
5 }
```

Event services do the real work of event processing. As they are modularly built, every event service should only fulfill a single purpose to keep the service lightweight. Whereas EventServer builds the infrastructure for event processing, everyone can implement their own event service. Just implement the interface shown in 5.2 and register it in the EventServer's IoC container.

Currently the following event services have been implemented. They differ in complexity and internal implementation, but due to their common interface and the IoC concept, more services could easily be added to the EventServer. Some of the event services are specific for the EventCloud applications. Others, like *Event2DBPersistence*, might be directly reused in other applications.

Event2DBPersistence stores the serialized event in a database.

EventRank1Indexing adds the given event to the Rank1 full-text index.

EventRank2BatchIndexing adds the given, correlated event to the Rank2 full-text index.

EventRank3BatchIndexing adds the given, indirect correlated event to the Rank3 full-text index.

ExecutionTimeMetricCalculation calculates a duration metric for a pair of correlated events.

LogPerformanceService is a service that calculates the event processing performance of the EventServer.

Multiple EventWorkers execute an event service at the same time. As event services are singletons in the EventServer container, they must be thread safe. From our experience, most event services can be implemented stateless because states are stored in the correlation session. Figure 5.7 in Section 5.3.7 shows an example of how correlation sessions are used to hold states.

5.3.6 System Service

System services are provided by the EventServer, offering services used by event services or other system services. System services are registered in the EventServer IoC container. Other services can obtain a reference to them via the container.

The **correlation service** is an essential system service for complex event processing. The event service API offers methods to call the correlation service. The correlation service encapsulates the whole event correlation complexity, as described in Chapter 3, from the event services which rely heavily on event correlation. Within the *.process* method of an event service, the defined correlations for the currently processed event can be fetched from the correlation service. When finishing the *.process*-method, all correlation sessions are returned to and persisted by the correlation service.

Listing 3.7 shows how event correlations are defined declaratively in the EventServer. These event correlations are accessible for the event services and can be handled by the correlation service at runtime. Event services request correlation sessions by the correlationset identifier and the correlation values of the currently processed

event. The correlation service has to handle the life cycle of the correlation session (create, retrieve, persist, invalidate).

Correlationsets (and their runtime representation correlation sessions) are globally available over all event services within the EventServer. That means that sessions are event-centered, not "event service"-centered. Sharing sessions over event services allows communication over event services borders via the session by storing relevant data into the session. As the event services for an event are processed one after another, the defined order of execution is of importance if the event services exchange information via the correlation session.

SENACTIVE InTime follows another approach where every event service has its own correlationset and so its own correlation sessions. Sessions are always tied to a single event service. Inter-eventservice communication cannot be done via sessions, it is done by creating new internal events and sending them to the following event services.

As discussed above, the correlation definitions are globally defined. An event service can request the session for a defined correlation from the correlation service during runtime. For example, the event service *ExecutionTimeMetricCalculation* can obtain the correlation session for *shipmentInfo*(Rank2), *transportInfo*(Rank2) and *AllOrderInfo*(bridged Correlation) when processing an event of the event type *eventtype://MediCare/TransportStart*.

Listing 5.3: ICorrelationService Interface

```
1 public interface ICorrelationService extends ISystemService
2 {
3
4     public Correlation session checkOutSession(String correlationSetIdentifer,
5       List<String> correlationValues);
6
7     public void checkInSession( Correlation session session) throws BaseException;
8 }
```

The interface of the correlation service is shown in Listing 5.3. The correlation service is not aware of events at all; it is sufficient to pass the *correlationset Iden-*

tifier and the event's *correlation values* to the service to obtain the correlation session. *checkOutSession* retuns the session to the caller. Internally this session is now blocked for other checkouts until *checkInSession* for this session is called. This is necessary to ensure that a correlation session is only modified by a single event at once. As it is possible that multiple event services with different events want to obtain the same session concurrently the correlation service has to block such calls until the first request has checked in the session again.

The correlation service implements monitors on session level to ensure correct concurrent behavior. As event services are highly parallelized the correlation service has to take care about the transactional behavior of correlation sessions. A detailed algorithm of the session retrieval from the correlation service is defined in Figure 10.1 in the appendix.

Other system services are available in the EventServer. SessionPersistence stores the correlation sessions into the EventServer database - a service the correlation service depends on. EventPersistenceService stores the serialized BaseEvent into the EventServer database.

5.3.7 Event Processing

To clarify the highly parallel way events are processed by the EventServer, refer to Figure 5.6.

In this example three, Events EventA, EventB and EventC - all of different event types - are processed concurrently. The events are received and transformed on the adapter. The EventDispatcher takes these events and assigns every event to a waiting EventWorker.

As all three events are of different event types, there are different event services subscribed that should be executed for each event:

EventA should execute the event services Event Database Persistence, Metric

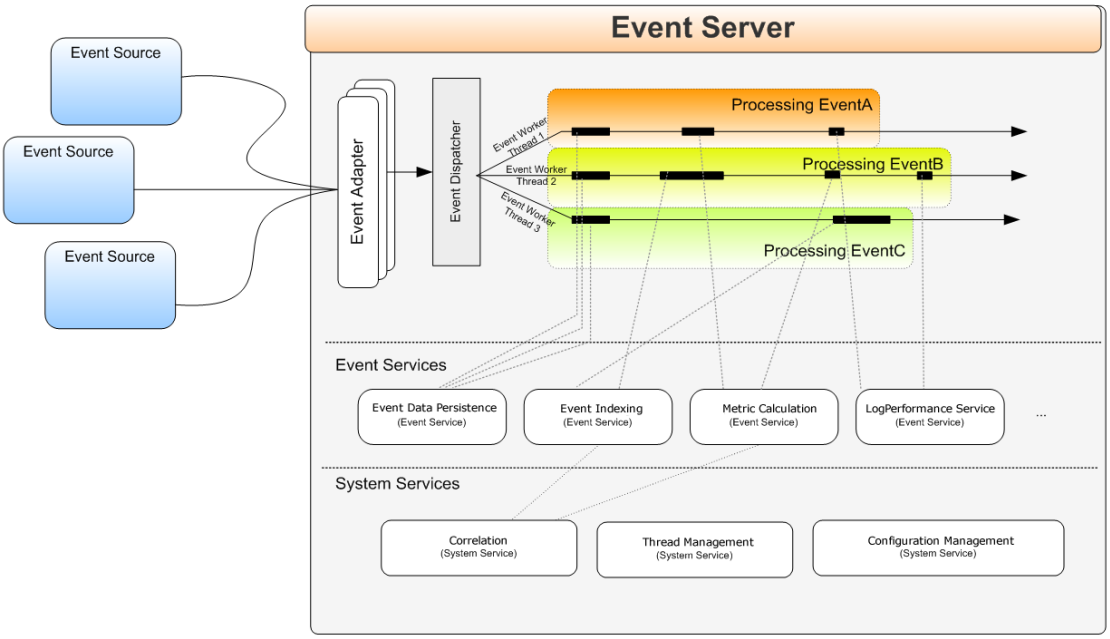

Figure 5.6: Parallel event processing in EventServer

Calculation and Log Performance Service.

EventB should execute the event services Event Database Persistence, Event Indexing, Metric Calculation and Log Performance Service.

EventC should execute the event services Event Database Performance Service, Event Indexing

As every EventWorker is running in its own thread, the scheduling depends on the JVM[6] and is not defined in advance. EventServer has to ensure that every event is processed independently from other events processed concurrently. The event services and the system services are singletons in the EventServer container and are accessed concurrently by EventWorkers.

The picture shows that event services are executed multiple times at once: the *process()*-method of *Event Database Persistence*e is executed in parallel by all three worker threads. Therefore, the developers of an event service have to ensure

[6] Java Virtual Machine

their implementation is thread safe.

The black bars illustrate that the execution time of an event service depends on what the service does. Whereas event indexing is a rather complex task, calculating a metric or logging the current throughput are services needing less computation time.

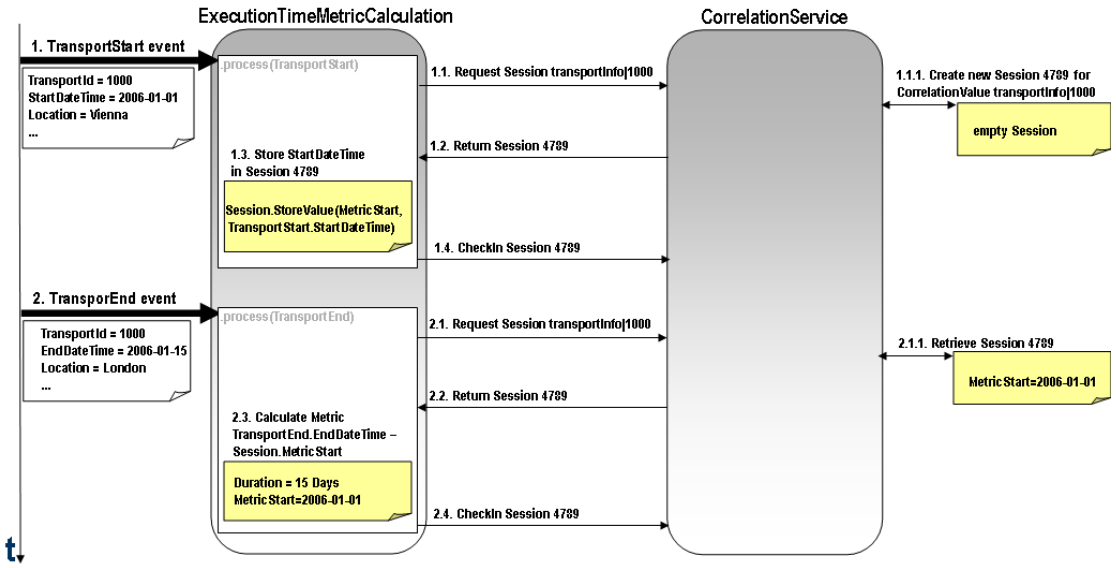

Figure 5.7: Event service and correlation service interaction

Some of the event services may need to use system services to execute their *process()*-method. Figure 5.7 shows how the *ExecutionTimeMetricCalculation* event service needs to call the correlation service to obtain the event's session to calculate the timespan between two correlated events. The figure describes the following example:

The service should calculate a metric **"Duration"** between the event types *TransportStart* and *TransportEnd*. These two event types are correlated by the correlationset *transportInfo* over the event attribute *TransportId*.

At time t1 a *TransportStart* event with *TransportId = 1000* is processed by the *ExecutionTimeMetricCalculation* event service. The event service calls the correlation service to retrieve the event's session for correlationset *transportInfo* with correlation value *1000*. As the correlation session does not exist, it is created by the correlation service with SessionId *4789*. The created session is returned to the *ExecutionTimeMetricCalculation* event service. The event service stores the event attribute *DateTime = 2006-01-01* of the *TransportStart* event under the key *MetricStart* in the session. The session is returned to the correlation service.

At time t2, an *TransportEnd* event with *TransportId = 1000* is executed by the service. The service calls the correlation service to retrieve the event's session for correlationset *transportInfo* with correlation value *1000*. The correlation service returns the existing correlation session with SessionId *4789* to the event service.

The *ExecutionTimeMetricCalculation* event service selects the *DateTime* attribute value of the currently processed *TransportEnd* event and retrieves the key *MetricStart* from the session. The event service calculates a duration of 15 days. It stores this metric under the key *Duration* in the session.

Other event services processing the TransportEnd event (or other services processing events that correlate with *transportInfo|1000*) can then retrieve the session and use the metric stored in the session to execute further logic.

5.3.8 Technologies

The implementation of EventServer relies on Java 5.0 and several mature open source projects. Java 5.0 was applied as it now includes a concurrency library in the namespace *java.util.concurrent* which was heavily used for the multi-threaded environment of EventDispatcher and EventWorker. The Spring Framework[7] was used as the Inversion of Control container for the EventServer configuration. All parts of the EventServer - EventAdapter, EventServices, SystemServices etc. - are configurable via the Spring configuration. To set up EventServer for scenarios

[7] http://www.springframework.org

other than EventCloud, one has only to modify the Spring configuration and their application-specific settings.

For the JMS-queue feeding the EventAdapter, two JMS implementations were tested: ActiveMQ[8] and Joram[9]. Both easily integrate with the JMS access provided with Spring 2.0. Apache Lucene plus the Compass API were used to create indexing event services for EventCloud's Rank1, Rank2 and Rank3.

5.3.9 Future Work: EventServer with Mule

Currently the EventServer is reimplemented with the help of the Enterprise Service Bus (ESB) Mule[10]. Relying on Mule offers additional adapters, event routing, high scalability, etc. out of the box and provides a mature platform for event processing. The concepts of event services, system services and event correlation are similarly used in this new architecture. Event processing is done with the help of the SEDA architecture[11].

5.4 EventCloud Frontend

The EventCloud Frontend was build by [28] and introduced in his thesis. I extended this implementation by adding user controlls for the novel concepts of Rank3 search and the metric view. EventCloud is a Google-like web frontend with search controls. An input box allows querying for (correlated) events. The result-set is displayed, ranked by relevance. Figure 5.8 shows the search options. *Over bridged correlationsets* searches in Rank3, *over correlationsets* in Rank2 and *only events* searches in Rank1. Per default all three indexes are searched, so if a user does not specify a searchrank, results of all indexes are returned. According to the definition, Rank1 results are listed before Rank2 and Rank3 results.

The original EventCloud frontend was extended to display metrics. As metrics

[8] http://www.activemq.org
[9] http://joram.objectweb.org/
[10] http://mule.mulesource.org/
[11] http://www.eecs.harvard.edu/ mdw/proj/seda/

Figure 5.8: Main search view

are calculated on top of correlations (see Chapter 7), the metrics are currently displayed along with each search result in Rank2 and Rank3. As this is a very simple approach, further efforts are necessary to integrate charts and diagrams for metric data.

Navigation between correlated events was introduced to simplify the task of analyzing and exploring events and their relation to each other. Section 8.4 shows an example how navigation is used to find complex coherences between events. Section 6.1 illustrates how navigation is done in the backend. For the EventCloud, frontend event navigation is simply realized via hyperlinks. If a specific event is selected, all correlations to this event are displayed on a new search result page.

The events are currently displayed in their XML representation, although this is not very user-friendly. Rozsnyai[28] built a first implementation of a pretty-print output module for events. A XSLT-approach was developed by SENACTIVE where the original event XML is transformed to a specific XML-tree format so a Javascript library allows to pretty print event types (Figure 5.9). Web technologies like JavaScript and AJAX allow editing of the original resultset so EventCloud becomes a more sophisticated user interface.

For the next steps, I think it is necessary to create alternative views on the event

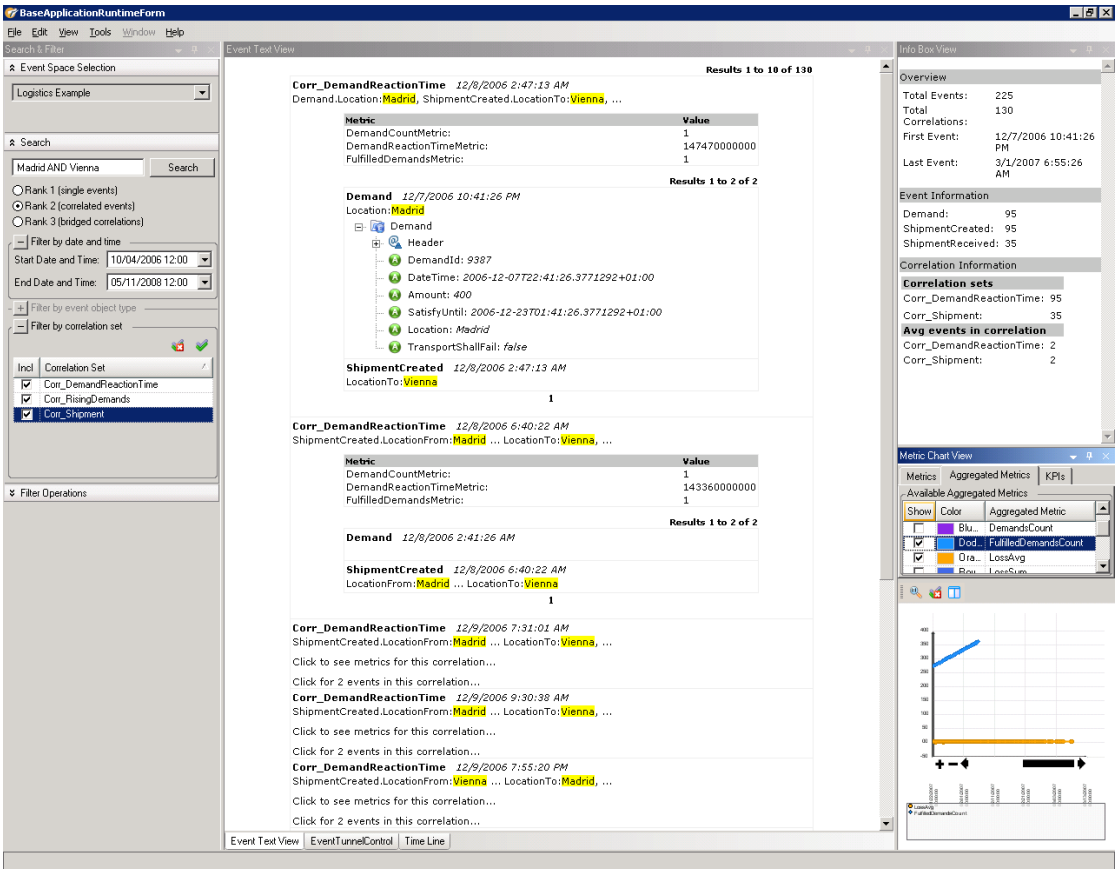

Figure 5.9: SENACTIVE EventAnalyzer: Textview [9]

resultset as complex event patterns are hard to grasp in the current text view. Information visualization offers a lot of knowledge to create visualizations for (correlated) event data.

Important aspects that visualizations must reflect for event data are:

Time aspect Latest events are often most relevant. So the time aspect, including interaction with the time like zooming or timewindow selection, is important to be handled by the visualization

Highlight event correlation In a cloud of events, it is essential to select and/or highlight correlation instances. As the visualization displays plenty of events, a user must be able to select specific ones.

Event Aggregation Visualization should collect the fine-grained events to create a higher aggregation level. Information visualizations like a *Newsmap* may be applied on events.

5.4.1 Frontend: Outlook to SENACTIVE EventAnalyzer

SENACTIVE is developing an application similar to the concepts of EventCloud. Figure 5.10 shows a graphical prototype of this tool which gives an outlook on future developments. Events are displayed in a novel information visualization called *Event Tunnel*. This approach allows the displaying of events, their properties, their correlation and their occurrence in time within a single view. The text view, currently implemented in EventCloud, has moved to the background and is just an alternative to the *Event Tunnel* visualization. Metrics are a more prominent feature with a separate chart box.

5.5 Performance

To validate the new EventCloud architecture, performance tests have been executed. Short tests with the old ETL-approach indicate that a comparison makes not much sense. Due to its implementation flaws, discussed in Section 5.1.3, the

Figure 5.10: SENACTIVE EventAnalyzer: graphical Prototype [9]

old implementation does not exceed the number of one-event-per-second for event indexing. Even the new approach has to handle event correlation, realtime event-processing (in opposite to the batch-insert used before) and a third full-text index for Rank3, we assume a clearly higher throughput.

EventServer was configured to process the following event services for each event type:

Event2DBPersistence stores the serialized event in a database.

EventRank1Indexing adds the given event to the Rank1 full-text index.

EventRank2BatchIndexing adds the given, correlated event to the Rank2 full-text index.

EventRank3BatchIndexing adds the given, indirect correlated event to the Rank3 full-text index.

ExecutionTimeMetricCalculation calculates a duration metric for a pair of correlated events.

LogPerformanceService used to log the performance of the test run.

For the performance test, an event simulation was created with SENACTIVE InTime Simulation Studio. The simulation generates 40000 Events of four different event types. EventServer was configured to correlate all four events in a bridged correlation and two of them in an additional direct correlation. The order of the events in the simulation causes session merges for every forth bridged correlation. Figure 5.11 shows the throughput for event indexing on a Pentium4 with 2GB RAM.

The throughput levels of at 19 events per second after 40000 events. This is a major improvement, and shows that the new architecture is a step forward. However, this value is not expected to be the limit of the presented approach. A Java profiler shows 15% of the time is spent in XPath expressions which are used to select event attributes needed for correlation. Investigating in a faster XPath alternative, such as another library or selecting event attributes by regular expressions,

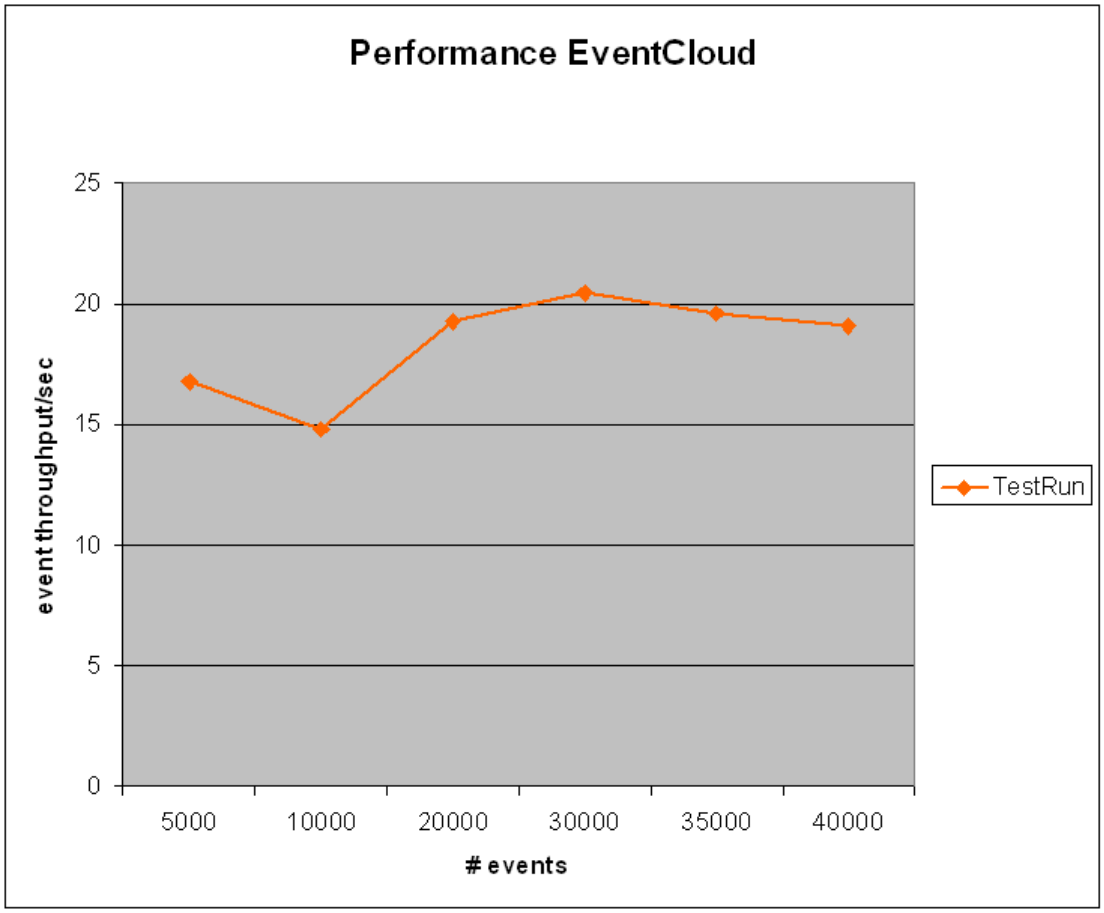

Figure 5.11: EventCloud performance test

should improve performance. Also, the performance test was more extreme than most real-world event scenarios. Because all events were published immediately, the correlation service has to block lots of correlation sessions as multiple events want to obtain the same session simultaneously. That's also the reason why the throughput for the first 10000 events is lower than those afterwards. In many real-world scenarios session locking is irrelevant because the time between correlating events, for example a *TransportStart* and a matching *TransportEnd* event, are at least minutes.

Search performance was not explicitly tested, because at the frontend EventCloud does not differ from any other Apache Lucene query application. The reader may visit [21] if they are interested in Lucene's search performance.

6 Indexing of Correlated Events

> *"At the heart of all search engines is the concept of indexing: processing the original data into a highly efficient cross-reference lookup in order to facilitate rapid searching."*, Gospodnetic, Hatcher [11]

As EventCloud is a tool to make events searchable, it is set in the area of information retrieval(IR). Many of the points important to IR are important for event search and analysis also, as discussed in Chapter 4. However, additional points arise from the characteristic of events. This chapter describes how the ideas of IR must be implemented for event indexing and searching. Chapter 4 has described a document oriented approach to store events for analysis, in the following chapter the implementation of this approach in EventCloud with the help of Apache Lucene is discussed.

Today, full-text search is a challenge often met in multiple contexts within the World Wide Web. These includes search-facilities on large websites, search engines or additional services, like Google's book search, or when there is a need to make a large number of other documents (like .pdf or .doc) searchable from desktop search applications on single workstations to business-wide solutions. For this wide area of requirements, numerous solutions, tools and APIs exist. A good overview about the tools available can be found in [27].

Full-text indexes were rather not used for analytical applications. This class of applications rely on the well-defined relational model and its variations, like the star schema used in OLAP. With EventCloud, we try to go another direction and implement complex analysis and ad-hoc queries on the basis of a full-text data representation. As described in [28], a relational representation of the search ranks

Rank1, Rank2 and Rank3 causes a major performance issue which directs us to the document-oriented approach, combined with event correlation, presented in this chapter.

Information retrieval is split into two major steps which are represented in Event Information Retrieval for event analysis as well:

Indexing Documents have to be intelligently indexed to make them searchable. So before a document is searchable, it must be added to the searchtool's index. Lots of research has been done to optimize the indexing process and structure. Full text indexes are often implemented as *inverted index* of the represented documents. The index holds the information about which term occurs in which document to allow a fast lookup. For a full-fledged index, advanced operations like stemming and stopword elimination must be applied during the indexing process. A good introduction on this topic can be found in [3] and [36]

Retrieval Documents should be easily found from the index again. Therefore, an intelligent result ranking and a powerful query language is essential. The query language is executed on the full-text index to return search results. To be able to restrict the resultset returned by a specific query, the query language must provide boolean operators like AND, OR and NOT, while sorting results by their relevance is done transparently for the user. As we learned from the history of the World Wide Web, a search engine experience is only as good as it's search results.

It should be clear that information retrieval is a complex topic. As it makes no sense to reinvent the wheel of information retrieval for event search, we did some exploration on existing tools which can help us to reach our goals with EventCloud. In the Java world there is an extraordinary API Apache Lucene [20], which

> *"provides Java-based indexing and search technology",*[20]

is an open source tool with a big community and was used in many different areas like search engines[1], desktop applications and webpages[2].

Nevertheless, be aware that making document searchable implies some different conditions than making events and their correlations searchable:

- The documents for indexing are large in comparison to events. The text of an .html-file will have several Kilobytes up to Megabytes. A single event has about 1 Kilobyte.
- Events are well structured and concentrate all information in a small amount of data. Documents are typically unstructured free text.
- The searchresult ranking algorithm is optimized for documents and take attributes like *position of search term in document* into account. So we have to question if this is relevant for events and returns meaningful result sets.
- Text documents exist before they get indexed, "event documents" in Rank2 and Rank3 are created on the fly to represent event correlation.
- Documents do not change that often. The content of a webpage may not change for a long time. In contrast, events occur irregularly in time at a high rate and the latest events are often the most valuable. To make the most recent events available in analysis the event index must be real-time.

We decided to use Lucene because of its numerous features we think are reusable for event indexing and searching with EventCloud. Because events, in their XML-representation, are not that different from ordinary text documents, Lucene offers an exciting toolkit to work with. To simplify the usage of Lucene and add additional features to the Lucene API, the Compass API [8] was used. Compass sets on top of Apache Lucene implementing extra features like transaction support, resource mapping and scheduled optimization.

[1] http://lucene.apache.org/nutch/
[2] http://wiki.apache.org/jakarta-Lucene/PoweredBy

6.1 How Correlation and Indexing Work Together

As we saw in Chapter 4, we do not only want to build a simple index on single events, but rather use the concept of correlation to create a searchable and navigatable **"cloud of events"** for event analysis. Therefore Chapter 4 defines the search levels Rank1, Rank2 and Rank3. With information retrieval, a user should have not only the possibility to **retrieve**, but also to **browse** the documents in the indexed document space. In our case, the documents representing events permit to browse correlated events that are within the same document and also to navigate to other, related events.

The following paragraph recaps how things work together:

For our event types, we have defined correlations in the EventServer. When an event is processed by the EventServer, it can activate the correlation session matching the attribute values used for correlation. We now could store[3] the whole event as XML text into the correlation session. Later, other events are processed which activate the same correlation session. All these events are stored to this particular correlation session.

This way a document is created, representing the events of one correlation instance, which should be searchable. For EventCloud, an event service was implemented which indexes the correlation instances of the EventServer. The index is updated every time a correlated event is processed.

The concept of event correlation is used to build **documents of interest** for Rank2 and Rank3 on the fly. The user declare, via their correlationset definitions, in which documents they are is interested for event search and analysis. The event service in the EventServer builds these documents by using EventServer's correlation service and adds the documents to the event full-text index.

[3] we do not really store the whole event in the session; this would blow up the session too much. We store the event only in the index

The following example defines a correlationset between *TransportStart* and *TransportEnd*:

Listing 6.1: Definition of the correlationset *transportInfo*

```
<correlationset identifier="transportInfo">
    <CorrelationTuple>
      <CorrelationData eventtypeUri="eventtype://MediCare/TransportStart">
        <XPathSelector>//TransportId</XPathSelector>
      </CorrelationData>
      <CorrelationData eventtypeUri="eventtype://MediCare/TransportEnd">
        <XPathSelector>//TransportId</XPathSelector>
      </CorrelationData>
    </CorrelationTuple>
</correlationset>
```

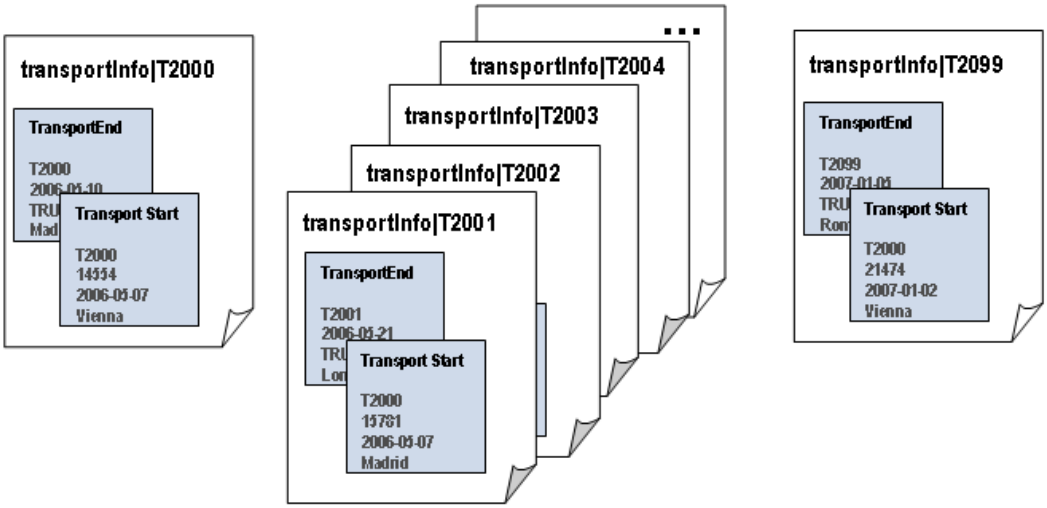

Figure 6.1: Documents in the full-text index

If 100 *TransportStart* events and 100 matching *TransportEnd* events are processed for EventCloud, the Rank2 index will contain 100 documents which are searchable. Every document is identified by the correlation value of the correlation session it represents. Each will hold two events with correlating values for the attribute *TransportId*. If a user now queries the index for some search terms, all document Ids which match the query will be **retrieved**. Figure 6.1 shows the 100 documents available in the index. A search for *Madrid* and *Vienna* will retrieve a subset of the documents in the index. This subset can be the start of a deeper event analysis.

As described in Chapter 7, additional data, like metrics, will also be stored along with the documents to support analysis.

Browsing or **navigating** through the events in EventCloud is an important feature to realize discovery analysis and exploration as described in Chapter 8. A user needs to navigate from a specific event to related events. As event correlation expresses a kind of *relation* between events, this concept is also capable for browsing.

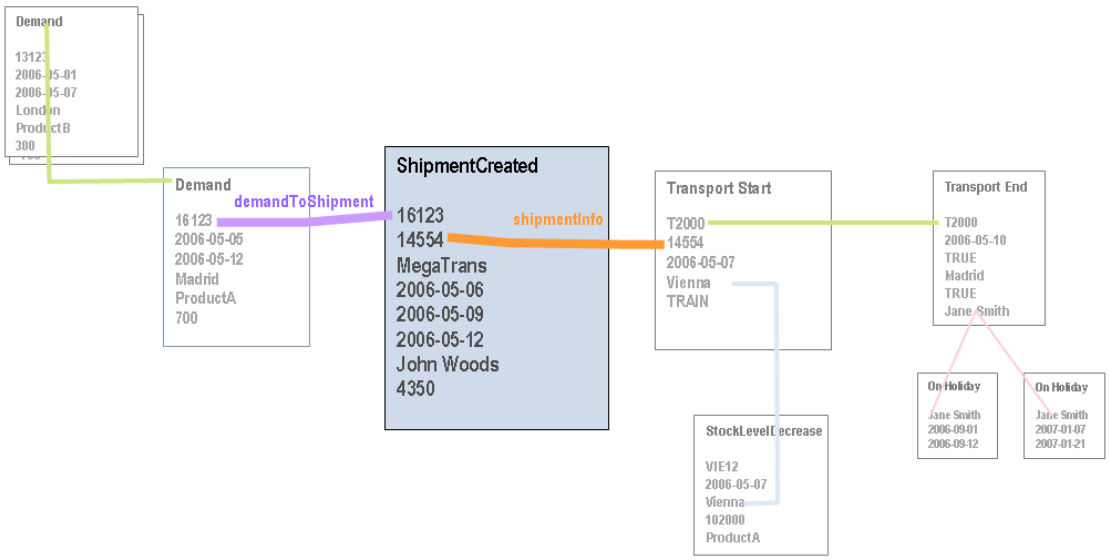

Figure 6.2: Browse events via correlations: ShipmentCreated event and its directly and indirectly related events

In Figure 6.2 a user retrieves a single *ShipmentCreated* event from the index through a Rank1 query. This event is the starting point for their analysis. To learn more about the event, the user wants to explore related events. The *ShipmentCreated* event participates on the correlations *shipmentInfo* and *demandToShipment*. Now the user can browse to other events that are related to the original *ShipmentCreated*, like the *TransportStart* event. The *TransportStart* event is again correlated with even more events. This way, a user has a facility to browse through the cloud of events along meaningful, defined paths of correlationsets.

Figure 6.3 shows the representation of the above example in the full-text in-

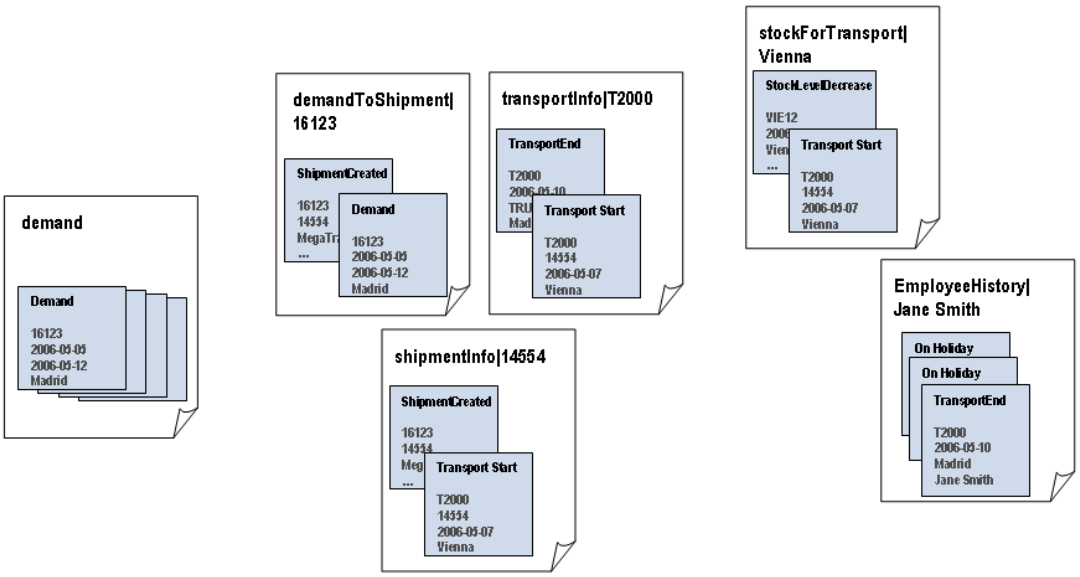

Figure 6.3: Index representation of the events and correlations defined in Figure 6.2

dex. Every correlation session is stored in a document. Navigation between events means navigation between the documents. This can easily be done with the Lucene query language, the event's identifier and the knowledge of which event type participates on which correlation. The selected *ShipmentCreated* event has an unique identifier, and with this identifier the index is queried to return all documents which include the event. In the figure's example, that would be the documents for *demandToShipment|16123* and *shipmentInfo|14554*. These documents include other events like the *TransportStart* event, that are directly correlated with the original *ShipmentCreated* event. By selecting the *TransportStart* event from the *shipment|14554* document, the navigation can lead to even more events like *TransportEnd* in *transportInfo|T2000*, and so forth.

The described approach allows to build a generic event search and analysis tool. A user has to define the event types which occur in their system and they have to define how they are correlated. The rest is done by the EventCloud system: it takes care of the correlation at runtime and it provides event services which are

creating and updating the search index. The EventCloud frontend allows event retrieval, browsing and analytical tasks.

6.2 Realization in EventCloud

As described in chapters 3 and 4, we have defined three levels of event search representing the different levels of correlation. The three Ranks are managed in independent full-text indexes:

Rank1 index a simple full-text index over all events that have been processed.

Rank2 index a full-text index over all correlationsets. A document in the index represents an instance of a correlationset in the EventServer.

Rank3 index a full-text index over all bridged correlations. A document in the index represents an instance of a bridged correlation in the EventServer.

Each index is managed by its own event service in the EventServer. The Ranks are separated into three indexes because each Rank needs different fields in the index and to allow search queries against a specific index. Every index will return different resultsets for the same search query, as shown in Chapter 4. Correlation not only allows navigation through the cloud of events, it also answers search queries that could not be answered by single events.

6.2.1 Rank1 Index

Listing 6.2 shows the Compass definition of the Lucene index for Rank1. A resource represents a document in the index. Every document has multiple fields (*resource-property*) which are stored along with the event.

Listing 6.2: Definition of a Rank1 index in Compass

```
1 <resource alias="event">
2   <resource-id name="guid"/>
3   <resource-property name="eventtype" store="yes" index="un_tokenized" />
4   <resource-property name="eventtext" store="yes" index="tokenized" />
5   <resource-property name="date" store="yes" index="un_tokenized" />
6 </resource>
```

For Rank1, the index is very simple to create. Every document in the index represents a single event that has been processed by the *Rank1Indexing* event service in the EventServer. Rank1IndexingService creates a new Lucene document every time the *.process()*-method is executed for an event. This document is added to the index as defined in Listing 6.2:

- Every document is identified by a *guid*. This guid is equal to the event's unique identifier.
- The event type of the event is stored in the field *eventtype*.
- The full-text of the event is stored in the field *eventtext*. Search queries are primarily executed against this document field, while the other fields store meta information of the event.
- The field *date* holds the date when the event was processed by the EventServer.

The *store=yes* attribute defines that the field is stored with the index. This means when a document is retrieved from the index, the document includes the full-text of the event, the event type and the date.

Another possibility would be to set *store=no*. This means the event's values are used during indexing to make the event's content searchable, but if the event is retrieved from the index, the fields are empty and only the resource identifier *guid* is returned. This way the full-text index will use less disk space, but the event must be persisted in some other storage for retrieval. To display an event as a result in EventCloud, the index would first be queried and then an additional round trip to the "event storage" would be necessary to display the event's content.

6.2.2 Rank2 and Rank3 Index

In Rank2 and Rank3, a document represents an instance of a correlationset. If an event causes the creation of a new correlation instance in the EventServer system, a corresponding new document is generated in the index storing the event's content. If a correlation instance is activated by a further processed event in the

EventServer, the corresponding document in the index is updated by appending the event's content.

Listing 6.3: Definition of a Rank2 or Rank3 index in EventCloud

```
<resource alias="correlatedEvents">
  <resource-id name="guid"/>
  <resource-property name="correlationname" store="yes" index="un_tokenized" />
  <resource-property name="correlationvalue" store="yes" index="tokenized" />
  <resource-property name="sessiontext" store="yes" index="tokenized" />
  <resource-property name="eventguids" store="yes" index="tokenized" />
  <resource-property name="updatedate" store="yes" index="un_tokenized" />
  <resource-property name="metrics" store="yes" index="tokenized" />
  <resource-property name="date" store="yes" index="tokenized" />
</resource>
```

The definition for the Rank2 and Rank3 indexes are equal. However they are separate indexes managed by separate event services for reasons discussed above. Although document definitions are equal, Rank2 stores different document instances than Rank3. Rank3 stores representations of bridged correlations whereas Rank2 only stores direct correlations. As we have seen in Figure 4.2, Rank2 and Rank3 can store the same events, but they are correlated in different ways. Section 4.2 had described what this means for search queries is:

- Each document is identified by a *guid*. This guid is equal to the correlation session identifier in the EventServer.
- The *correlationname* is the name of the correlationset this document represents. e.g. *transportInfo* or *AllOrderInformation*
- The *correlationvalue* stores the attribute values for the correlation instance this document represents. e.g. *TransportId=T2000* for the correlationset *transportInfo*
- All events that have participated on the correlation instance are stored in the field *sessiontext*. This field has the fulltext of all events of the represented correlation instance. Search queries are primarily executed against this document field, while the other fields store meta information of the correlation.
- *eventguids* stores all event identifiers. This is done for optimization purposes.
- *updatedate* stores the date this document was last updated.

- *metrics*: if additional metrics were stored in the correlation session, these values are stored in this field. A metric is stored in the format *METRIC-NAME=VALUE*. Multiple metrics are stored as a comma separated list in this field. This approach has two advantages: *(1)*The metric is stored along with the Lucene document. If the document is returned as a searchresult, we immediately have all metric values to this correlation instance without a further roundtrip to a database *(2)*metricnames and metricvalues are also searchable for a user through the Lucene query language.
- *date* stores all dates of the events in this correlation instance. That's necessary to allow date-range queries with Lucene.

While the Rank1 index is straight forward, the index for Rank2 and Rank3 are more complex to create. Rank1 index is an index where documents are only added. Every new event generates a new document which is put into the index. Because no correlation comes into play, Rank1 is easily implemented with Lucene.

The index for Rank2 and Rank3 must update existing documents in the index as event correlation is mapped into the index. Every event processed in the EventServer may relate to previous and future events according to the event correlation definition. For each of these correlations, the event activates the matching correlation session in EventServer and the equivalent "document of interest" in the index needs to be updated.

The algorithm for Rank2 index event service can be described as:

1. An EventWorker starts executing the *process()*-method of the Rank2Indexing event service for EventA.
2. For each correlationset defined on EventA's event type the correlation service is called to return the matching correlation session.
   a) For the returned correlation session the index is queried.
   b) If a document for this correlation session already stored in the index, it is retrieved. Otherwise a new document is created.

c) The current event is added to the document. The document including the new event is indexed and added to the index again.

3. The EventWorker finishes executing the Rank2 indexing event service

Rank3 indexing needs to manage *SessionMerge*. As described in Section 3.3 a bridged correlation can cause multiple sessions to be merged during runtime. This session merge is transparently handled by the correlation service. The information of a merge must also be forwarded to the Rank3 indexing service to update the documents in the index according to the session merge.

If a merge needs to happen in the EventServer because of the currently processed event, the indexing service has to reflect this merge in the Rank3 index, too. The old documents representing the merged session must be removed and the merged session needs to be updated in the index. The current implementation of the EventServer correlation service uses the correlation session to signalize the indexing service that a merge has occurred. On a *SessionMerge*, the correlation service sets a flag and a list of the merged correlation sessions in the correlation session that is returned to the Rank3 indexing service. This returned correlation session is the "winner" of the merge. The list of merged correlation sessions in the returned correlation session is no longer valid in the EventServer. These correlation sessions have been merged to the "winner" and removed in the EventServer. For the Rank3 indexing service, this means a session merge always causes multiple documents in the index to be removed and a single document (that represents the removed documents plus the event that causes the merge) to be added.

One might ask why we care about this special case. Wouldn't it be sufficient to wait until all events of a correlation have been processed and add them afterwards to the index. We say no: as soon as an event is processed by the EventServer, it should be available in the search index of EventCloud. The index should be as up to date as possible. Waiting for all events of a correlation can take hours, up to days[4], or is not possible at all[5].

[4]think about *TransportStart* and *TransportEnd* event on an transport between Vienna and London
[5]If you correlate all *Demand* events there is never a "last" event

It is an EventCloud mantra to index an event as soon as it occurs, even if additional complexity arises from this circumstance. The added value of real-time data in an analytical tool like EventCloud is beyond the costs of handling this complexity.

The algorithm for Rank3 index event service can be described as:

1. An EventWorker starts executing the *process()*-method of the *Rank3Indexing* event service for EventA.
2. For each correlationset defined on EventA's event type, the correlation service is called to return the matching correlation session.
   a) if a *SessionMerge* has happened, retrieve all documents representing the correlation sessions merged. Add the values of all documents to the document that has won the merge. Remove the documents from the index because they no longer represent a correlation session in the system.
   b) if no *SessionMerge* has happened, retrieve the document for this correlation session if it exists, otherwise create a new document.
   c) The current event is added to the document. The document including the new event is indexed and added to the index again.
3. The EventWorker finishes executing the Rank3 indexing event service.

Figure 6.4 shows a document merge in the Rank3 index. The initial situation in this example is the bridged correlation *AllOrderInformation* defined in Listing 3.7. At **time 1)**, the *Demand* event as well as the *TransportStart* and the *TransportEnd* event have already been processed and added to the Rank3 index, so two independent documents exist in the Rank3 index. At **time 2)**, the *ShipmentCreated* event is processed in the EventServer. According to the definition of the bridged Correlation *AllOrderInformation*, the correlation service has to execute a *SessionMerge*. This merge is propagated to the Rank3Indexing event service to update its index correctly according to the *SessionMerge*.

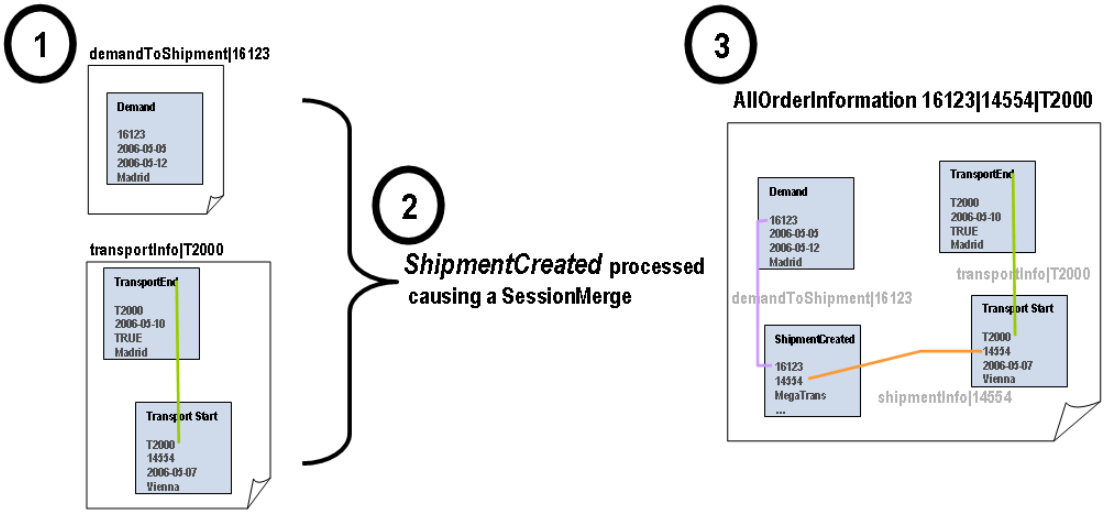

Figure 6.4: *Document merge* in the Rank3 index after a *session merge* caused by *ShipmentCreated* occurred in EventServer

The *Rank3Indexing* service has to update its document structure now. The two documents existing at time **time 1)** need to be merged into a single document representing the bridged correlation *AllOrderInformation 16123|14554|T2000*. **Time 3)** shows the updated index after the document merge. A single document exists in the index, holding all four events processed.

6.3 Shortcomings using Lucene

As Lucene covers a lot of background tasks with indexing and searching events, it also has some shortcomings when used with real time event processing. This section points out the problems identified during developing the indexing event services with Lucene.

6.3.1 Missing Update-support for Index Documents

Lucene does not support real *Update*-statements. It only allows *Delete-and-Add* operations. This drawback needs to be handled when developing a Rank2 or Rank3 index.

Because every new event may participate on an existing correlation instance, EventCloud permanently has to update the corresponding documents in the index to provide a real-time index. As no update operation is available, the document must be retrieved from the index, the single event added, the old document removed from the index, and the new document has to be completely re-indexed to be added to the index.

Re-indexing the complete document just because a single event has been added to a correlation is dissatisfying. Imagine a correlation session storing thousands of events. Adding a single event to this session causes the whole document to be retrieved, deleted and re-indexed. This could be a major bottleneck.

A good solution for us would be an *Append*-method in Lucene. The documents in Rank2 and Rank3 do not change; rather they expand with each additional event. A new event is appended to the existing "document of events". This means Lucene would not have to re-index the complete document, but only index the newly added event. However it would need some investigation to check if this feature is possible to implement in Lucene. Currently we are living with this restriction because Lucene's indexing performance is fast enough for the implemented scenarios. However on big documents (i.e. correlating many events in a single session) and high throughput of events this problem must be kept in mind.

6.3.2 Issues with Real-time Indexing

Lucene allows thread-safe adding of documents to the index. The synchronization is realized via a file lock that manages access to the index. Before a thread is allowed adding data to the index it has to obtain the file lock. After finishing, it releases the file lock.

When I developed the Rank2 and Rank3 indexes, I first made the naive assumption that every thread immediately adds events to the index with it's own IndexWriter instance. When testing the services, *TimeOutExceptions* were thrown by Lucene

and performance was down. I realized that the multiple threads were busy obtaining and releasing the file lock and creating an IndexWriter instance but do little work indexing documents. At some point all threads were waiting for the file lock, until *TimeOutExceptions* were thrown after 60 seconds.

I rethought my design and was forced to use a single thread with a single *IndexWriter* to add events to the index. The indexing event service collects multiple events for indexing when an indexing thread is started to update the index. Nevertheless, this was a fundamental drawback from our goal to reach real-time event indexing.

To minimize the latency, the following strategies may be applied for starting the indexing thread:

Number of events indexing thread is always started after a configurable number of events processed. Setting this value to 1 will lead to real time event indexing, however this can cause problems on heavy load. Setting the value to a high number will cause some kind of batch indexing.[6]

Time indexing thread is always started after a configurable timespan. So even if only one event has been processed during the defined timespan it can be guaranteed that this event is added to the index

Define on event type level : As different event type levels are more or less important to be added to the index, it may be interesting to define the above strategies on event type level. If eventtypeA must be available in the index immediately, one might set "Number of events" to 1 for this event type. For eventtypeB, it could sufficient to be added to the index within 10 minutes, so "Time" is set to 600 seconds.

[6] A higher value potentially allows some optimization. If multiple events update the same document this can be done at once, instead of doing the update-steps multiple times on the same document

6.4 Further Topics on Indexing

Because event indexing and search is essential for EventCloud, I would like to address some further topics that may need additional research. These topics could be the input for further work on indexing events and are particularly important for everyone developing event indexing and searching in a large scale environment.

6.4.1 Index Location

Currently the indexes are stored in the file system. However, Lucene allows to transparently change the location of the index. Existing implementations allow a Lucene index to be stored in:

RAM the index is stored in the RAM. Indexing and searching are very fast but the index is not persisted.

FileSystem the index is stored in the file system. That's what EventCloud currently implements.

Database the index is stored in a JDBCDirectory.

I made some simple benchmarks and compared them with some figures from Lucene's performance tests [21].

```
- Event as simple XML-Stream with StandardAnalyzer
- .optimize() called once at the end of indexing
- IndexWriter open()/close() every 100 events to flush results to index
- Rank1 simple indexing
- 5400 events


FILESYSTEM(FSDirectory) => 100 events/second
RAM (RAM Directory) => 360 events/Second
```

As expected, RAMDirectory is much faster than a filesystem index. Unfortunately it's not very useful because EventCloud needs the index to be stored persistently

somewhere. For our requirements, FSDirectory seems fast enough, however someone might find scenarios where this is not sufficient. The following ideas should be considered:

- Use InMemory databases like HSQLDB[7]. HSQLDB is able to handle data up to 8GB in main memory. The database can also be persisted to the filesystem. This way, the speed of the RAMDirectory with the safety of the FSDirectory could be combined.
- Use RAMDirectory as buffer. Lucene can add an index to an existing index. RAMDirectory could be used as temporary buffer to add this index later to the main index stored somewhere in the filesystem. Lucene internally performs a similar action with the parameter *"mergeFactor"*.
- Make sure that the indexing location is really a bottleneck for the application. We experienced that other constructs cost more performance than indexing.

As indexing is just one part of of information retrieval, you have also think about search performance. The case studies in [11] asserts that searching 4 million documents can be done in < 100 milliseconds with Lucene.

6.4.2 Optimization

To optimize search performance, Lucene offers an *optimize()* method. With the first version of EventCloud some confusion arose with the usage of this method. Following is a clarification, along with ideas about when to call this method for a frequently updated full-text index.

To understand *optimize()* it is suggested to read through [11]. Some interesting points to mention:

> *Adding new documents to an unoptimized index is as fast as adding them to an optimized index.*

[7] http://www.hsqldb.org/

It's important to emphasize that optimizing an index only affects the speed of searches against that index, and doesn't affect the speed of indexing

optimize() DOUBLES disk space requirements

[11] explains that *optimize()* only increases the speed of search queries against the index. Calling *optimize()* causes Lucene to create a copy of the original index in the background, so the space requirement is temporarily doubled. Lucene optimizes this copy and then replaces the original index to release the space.

With traditional batch indexing jobs, this is not a big problem as long as enough disk space (or for RAMDirectoy memory space) is available to optimize the index. The ideal moment to call *optimize()* will probably be at the end of the batch indexing job. As the index won't change until the next batch job, it is reasonable to optimize the index.

With indexing events we do not have the ideal point in time. Events occur irregularly in time and therefore are added irregularly to the index. Fortunately, indexing speed is not affected by a fragmented index. However while the index is optimized, no new events can be added to the index. In this way, *optimize()* causes some problems with the goal of real time event indexing.

So what I want to point out is, that the optimize() method has to be used carefully. It should only be called, if the performance of the search queries is really decreasing. Optionally, Compass offers a *ScheduleOptimizer* that can be scheduled to run at specific time windows, for example every Monday at 00:00, which may be a good approach if the events occur only during business hours. For other scenarios it is rather hard to grasp when to call *optimize()*.

6.4.3 Boosting Strategies

Every document in a Lucene index can have an individual boost factor set. This boost factor is used for Lucene's result ranking calculation. So this boost factor

allows ranking a document higher in the resultset which otherwise would not be on top.

As documents are events, or correlated events in our case, we could think about strategies on how to apply *boost* on event documents:

Time boosting Recent events (or correlations) are often more interesting than older ones. To ensure that recent events are on top of the search result, one can boost these events or reduce the boost of older events.

Type of Document Every document in the index is of a specific event type (or a specific correlationset). A boost can be applied to a specific event type so this type is always on top of the search result.

Content of the Document A more elaborate approach would be to look into an event to set its boost factor. Maybe it is reasonable to add a boost if a specific event attribute is set or, for correlations, if the number of correlated events is greater than X, or if a metric has the value Y. This approach depends on the concrete scenario.

Unfortunately, the boost factor cannot help in every situation. A boost factor is defined on document level (or field level) in Lucene. However if different users are interested in different event types or correlationsets, the boost factor is not adequate. As an alternative, a *boost factor strategy* based on a user's profile could be introduced.

6.4.4 Distributed Indexing Strategies

As a search tool grows, it may need to be distributed and the indexes may need to be split. Currently, the EventServer and EventCloud with its indexing services run on a single node creating a separate index for each Rank1, Rank2 and Rank3. Search queries are executed by the same node against the indexes.

If a high indexing performance is necessary, EventCloud would have to balance indexing on multiple machines. A simple solution discussed with Lucene is to

store the index on a network share and make it accessible by all nodes. Another option would be to split the index in multiple parts on different nodes, where every index represents a certain part of the full index. These options provide more scalability, but many problems known from distributed systems arise. For a user, it should be transparent whether they access a single index or a distributed one. A search query executed must always return the same result, independent from the actual index representation.

It is not simple to run Lucene as a distributed service. Good discussions and possible solutions can be found on the Lucene mailing list[8].

[3] points out two approaches to split inverted index files which come from the internal structure of inverted files. *Document partitioning* slices the documents into multiple subindexes. The query must be executed against each subindex. *Term partitioning* slices the indexing terms. So the query evaluation procedure of each document is distributed. With the Rank indexes in the EventCloud system, we can even think of different approaches to split the index into subindexes. The following paragraph gives some suggestions to create proper subindexes.

Approaches for Rank1:

No splitting Currently implemented. All events are stored in a single index.

event type Index is split into subindexes for every event type. One node is responsible for one event type. This can be helpful if a specific event type needs much more performance than other types. This event type can be processed by the fastest machine.

Time Index is split into chunks of time. Every index only stores the events of a specific time period. After a defined time, a new subindex is created.

Approaches for Rank2 and Rank3:

No splitting Currently implemented. All correlations are stored in a single index.

[8] http://www.mail-archive.com/Lucene-user@jakarta.apache.org/msg12700.html

Correlationset Index is split into subindexes for every correlationset. One node is responsible for one correlationset. This can be helpful if a specific correlationset needs much more performance than other sets. This correlationset can be processed by the fastest machine.

Time Index is split into chunks of time. Every index only stores the correlated events of a specific time period. After a defined time, a new subindex is created. This approach is rather complicated because some correlation sessions can last for a long period of time. A strategy would be need to handle the documents representing these correlations.

Generational Collector

Another more advanced approach is similar to generational garbage collectors introduced by [18]. Every document in a Lucene index is assigned to a generation. A subindex stores all documents for one generation. Some kind of "Document Collector" is started from time to time to move a document to another generation, based on it's last activation time.

This approach is particularly interesting for correlationset indexing in Rank2 and Rank3. A correlation session is activated by any number of events. Maybe these events occur in a short period of time or maybe they occur over a long timespan. For indexing, this means every document in the index may be updated by the next event. However, it is quite plausible that a document which has not been updated for a long period, say one month, will no longer be updated. So an approach would be to define generations for documents.

Generation 1 A small set of documents representing correlations which have been activated recently. This includes all new documents.

Generation 2 A larger set of documents representing correlations which have been activated in the last X minutes.

Generation 3 The largest set of documents representing correlations which have not been activated within the last X minutes.

All documents of a generation are stored in their own subindex. This permits defining a different location for every index as discussed in Section 6.4.1. Documents which are updated frequently (Generation 1) may be stored in a RAMDirectory, whereas documents which have not changed for a long time are stored in a file-based directory.

7 Metrics for Correlated Events

With the first version of EventCloud a tool was built to search over correlated events in the first place. However, when we were using the tool, we realized that for analytical tasks it is necessary to retrieve metrics along with the event resultset and display them immediately in the EventCloud frontend. As events include detailed information on the activities they represent, this information is of limited interest for many usage scenarios. A human user working with our tool needs to see interesting information immediately, not hidden somewhere in events (or even scattered over multiple events).

7.1 Metrics with EventServer and EventCloud

With this version of EventCloud, we developed the first approach to integrate metric data into the event search. The user has the facility to use the EventServer to implement event services that calculate the metrics which are interesting for their scenario. When the events are processed by the EventServer, before they get indexed by the Rank1-, Rank2- and Rank3- indexingservices, other event services calculate metrics whose results are then added to the index along with the event. The event service subscription functionality of the EventServer is used to enrich the events (or more exact the correlation instances) with metrics data before they are indexed.

Figure 7.1 shows how metric calculation and indexing interact. In the *ExecutionTimeMetricCalculation* event service, the *Duration* metric is calculated as shown in Figure 5.7. The correlation session stores the metric *Duration*. Because *EventRank2Indexing* event service activates the same correlation session when process-

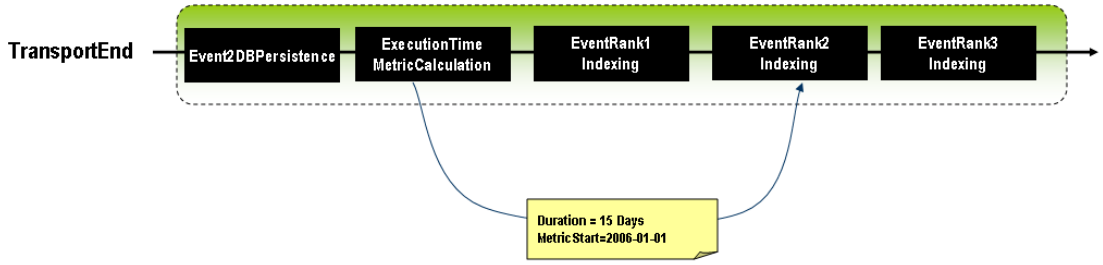

Figure 7.1: Execution of the subscribed event services for a *TransportEnd* event. Metrics must be calculated before the execution of the indexing service to be available in the full-text index.

ing the *TransportEnd* event, it can access the *Duration* metric in the session and add it to the full-text index.

Metrics in EventCloud are typically defined on top of correlationsets. Correlationsets are declared to collect events that are related to each other. By defining a correlationset in the EventServer, a user creates their own *"eventstream of interest"* within the stream of events processed by the EventServer (Figure 3.1). Events that are correlated during runtime (that activate the same correlation session) establish the input values for a metric calculation of a correlation instance.

As compared to the first version of EventCloud which only used correlation to create searchranks and to make the events navigatable in search, metric calculation is a second application for event correlation. This should underline the significance of event correlation. It is an essential concept facilitating complex event processing.

Typical metrics which can be defined with correlated events are:

Duration The timespan between correlated events matters in many ways. We have seen this metric in the first example on correlation in Chapter 3, as well as in Figure 5.7.

Average value The average value of an event attribute over all events in a corre-

lation instance. Typically this can be some kind of cost metric.

Maximal and minimal values The maximal or minimal value of an event attribute over all events in the correlation instance.

Deviation The deviation of an event attributes from a given value. Interesting, for example, in stock price calculations.

With the concept of correlation sessions and event services in EventServer it is straight forward to calculate a typical metric like *Duration* for a correlation instance:

1. Define a declarative correlationset, correlating the *starting event*- and the *end event*-type
2. Implement an event service:
   a) when the *starting event* is processed: request the correlation session, select the *DateTime* attribute of the event and store the value into the correlation session
   b) when the *end event* is processed: request the correlation session, select the *DateTime* value from the correlation session and subtract it from the *end event*'s *DateTime*. The difference is the metric *Duration* and can be stored in the correlation session or published to some other system.

The correlation session is used as temporary storage for the metric calculation. The attribute values from the events which are necessary for the metric calculation are stored in the session until all values are available to compute the metric. Duration metrics typically have a start- and an end-event. The metric is finally calculated when the end-event is processed. Before that moment, the metric is not available for this correlation instance. Average metrics are typically available with the first event that activates a correlation session. Every additional event which activates the correlation session updates the metric.

By now we have seen how to execute metrics on the level of correlationsets. For every correlation instance of a correlationset, the metric is calculated. Since metrics

Single correlation instance *Level 1*	Resultset *Level 2*	Defined correlationset *Level 3*
Duration for a specific correlation instance *transportInfo=100*	Average Duration over all transports in a resultset	Average Duration on all correlation instances of correlationset *transportInfo*

Table 7.1: Levels where metrics are applied in EventCloud

should be meaningfully displayed in the EventCloud frontend, additional requirements exist. When a correlation instance is in the result of a search query, we can simply display all metrics which were calculated for it. However, when a user performs search queries, they are also interested in the average value of a metric within the current resultset returned by their query. An example would be the **average duration** for all transports which go from *"Vienna to London"*. Another query needs to know the average duration of all transports from *"Vienna to London"* with carrier *"Mechatrans"* delivering *"ProductA"*. Because different correlation instances are returned for each of these queries, the *average duration* metric will differ for each query.

As a user can execute complex queries that return different resultsets, it is impossible to calculate all possible average metrics in advance. The EventCloud frontend needs to calculate these metrics on the fly depending on the resultset of the executed search query. Three different levels have been detected where metrics have to be applied in EventCloud. Table 7.1 shows these levels.

Single correlation instance(Level1) was discussed at the beginning of this chapter. As we have seen, this metric can be calculated during event processing by event services with the help of event correlation. *Resultset(Level2)* are metrics that EventCloud must calculate on the fly from the resultset of a given search query. *Defined correlationset(Level3)* is the overall metric for all correlation instances of the defined correlationset. This metric differs from "Metric calculated from resultset", except the search query has returned all correlation instances of a correlationset.

correlation session	From-To	Duration
transportInfo=17	Vienna-London	16
transportInfo=25	Vienna-London	20
transportInfo=37	Munich-Vienna	8
transportInfo=45	Vienna-London	17

Table 7.2: Simplified EventCloud index with metric *Duration*

Single correlation instance	Resultset	Defined correlationset
For *transportInfo=17*: 16h; For *transportInfo=25*: 20h; For *transportInfo=45*: 17h	Average for three matches in resultset: 17,67h	Average over all *transportInfo* in the index: 15,25h

Table 7.3: Metrics returned on metric levels for search query *"Vienna and London"*

Let's make an example for this definition. Table 7.2 shows a simplified full-text index for the correlation *transportInfo* and for each transport the calculated metric *Duration*. Executing the search query *"Vienna and London"* against the index returns the metric data as shown in Table 7.3 for the three levels defined.

In this particular example, the overall metric for the correlationset *transportInfo* makes little sense for a user. The average duration over all *transportInfo*-instances is not significant because all transport between all locations are aggregated, averaging transport durations between different locations. However, it may be useful for other scenarios. Calculating the overall metric over a correlationset is currently done by implementing an event service. This service sums all metrics calculated in correlation instances of a specific correlationset.

7.2 Implementation with Lucene

As described in Section 6.2.2, metrics are stored along with the correlated events in the full-text index. If metrics are calculated for a correlation instance, the corresponding document in the index stores these metrics in a comma separated key-value list (Figure 7.2).

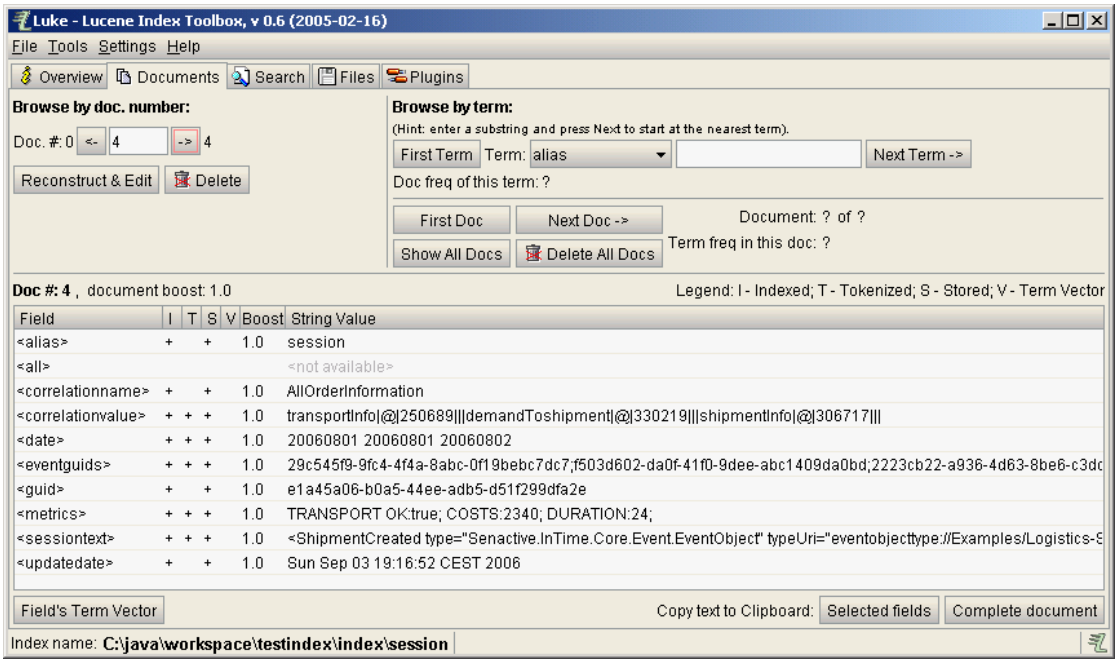

Figure 7.2: Insight into the Lucene Rank3 index. Metrics are stored in the index field <*metrics*>

If a single document is retrieved from the index, all metric data for this instance is available immediately for display in the EventCloud frontend. Because of this, and the fact that metrics are searchable when they are added to the index this way, we first thought this is a very elegant solution for storing metrics. However, problems arise with the calculation of average metrics in the resultset.

Lucene calculates an internal **hit-list** holding the identifiers of the documents which are returned by the search query. The documents are only loaded into main memory if one explicitly retrieves them from the index. Because searchtools normally only show a list of 10 results per page, only 10 documents are loaded from the index at once. In our case, to calculate an average metric over all documents in the resultset, all documents (with Lucene you can't retrieve a single field from a document) need to be retrieved just to get the metric values. We experienced that Lucene performs poorly if hundreds of documents have to be loaded into the main memory. The calculation of the Level2 metric lasts for several seconds.

We did not make a comparison, but storing the metric data of a correlation instance into a database would probably perform better. This could be easily done with an additional event service. For further research, a better solution should be found.

7.3 Limitation of Predefined Metrics

Metric calculation for events is done during event processing before the full-text index is created. For a user of the EventCloud frontend, these metrics are displayed when executing search queries.

In comparison to OLAP, the metrics needs to be defined before event processing as they cannot be applied easily afterwards. With OLAP, new metrics can always be defined on the data cube. So an analyst can add or modify metrics while looking at the same data set. Even if all event data is available in the indexes, there is currently no way to create new metrics with EventCloud after event processing is done. All relevant events would need to be processed again by the EventServer with the new metric definitions. This action would take time and reconfiguration of the EventServer it's not an adequate solution for this problem.

The necessity of predefined metrics is a significant limitation of the current metric implementation for EventCloud. Further research is needed in this area to find a more practical solution.

7.4 Declarative Metrics on Correlationsets

As we implemented multiple scenarios with EventCloud, we learned that many metrics recur in every scenario. As we have implemented event services to calculate the metrics, we made them configurable to potentially reuse these services in multiple scenarios. Because of EventServer's configurable architecture, metric event services can be plugged to the server as needed by a concrete scenario.

We also learned that this approach is not very handy and is error-prone. A better approach would be to define the most common metrics for correlationsets with an XML declaration. Instead of implementing individual event services which need to be registered in the event service subscription, a system service *Metric Calculation Service* calculates metrics at the correlation session checkout.

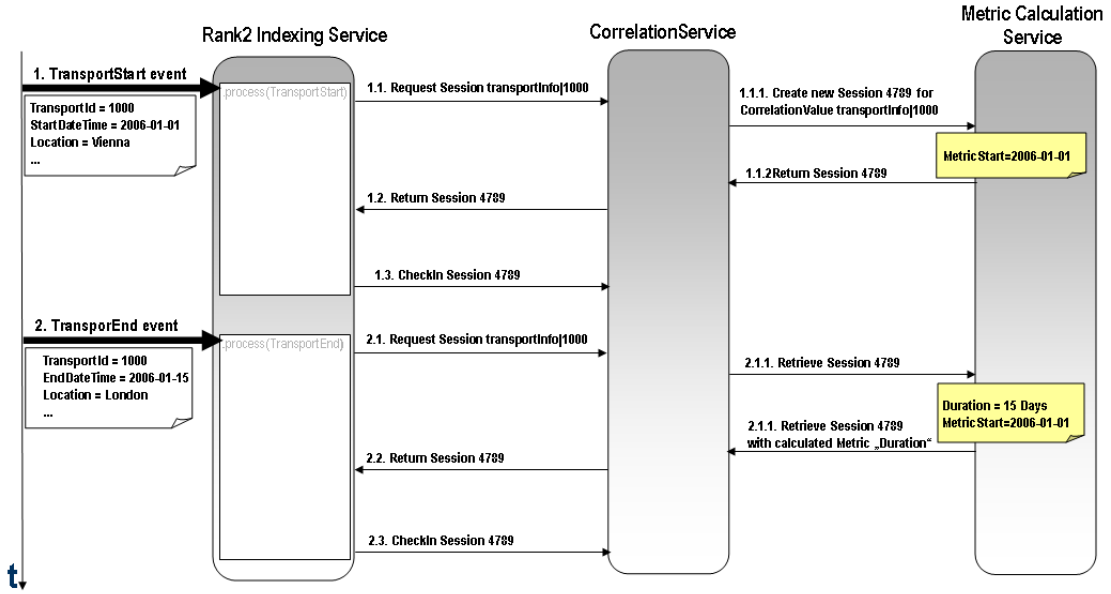

Figure 7.3: The Metric Calculation Service calculates the *Duration* metric during session checkout

Figure 7.3 shows the calculation of the *Duration* metric with the same result as in Figure 5.7 but with the help of the metric calculation service. In Figure 5.7, an extra event service was implemented to calculate the *Duration* metric, although metric calculation is only a preparation step for event indexing as described in Figure 7.1. With this new approach, one can immediately subscribe to the Rank2

Indexing Service because metric calculation is done in the background during correlation checkout. A user has to define which metrics they are interested in, then the values are transparently calculated by the metric calculation service. As a result the number of event services in the event service subscription list should be reduced and focused to those services needed to solve the event processing tasks in the concrete scenario (like indexing for EventCloud).

Figure 7.4 shows a metric definition prototype. With this XML schema it is possible to define metrics declaratively, while at runtime the metric calculation service uses this definition to actually calculate the metrics. Every metric is identified by its *identifier* and references a single correlationset via *correlationset identifier*. A metric always returns a single value for the calculation given in the tag <*Calculation*>.

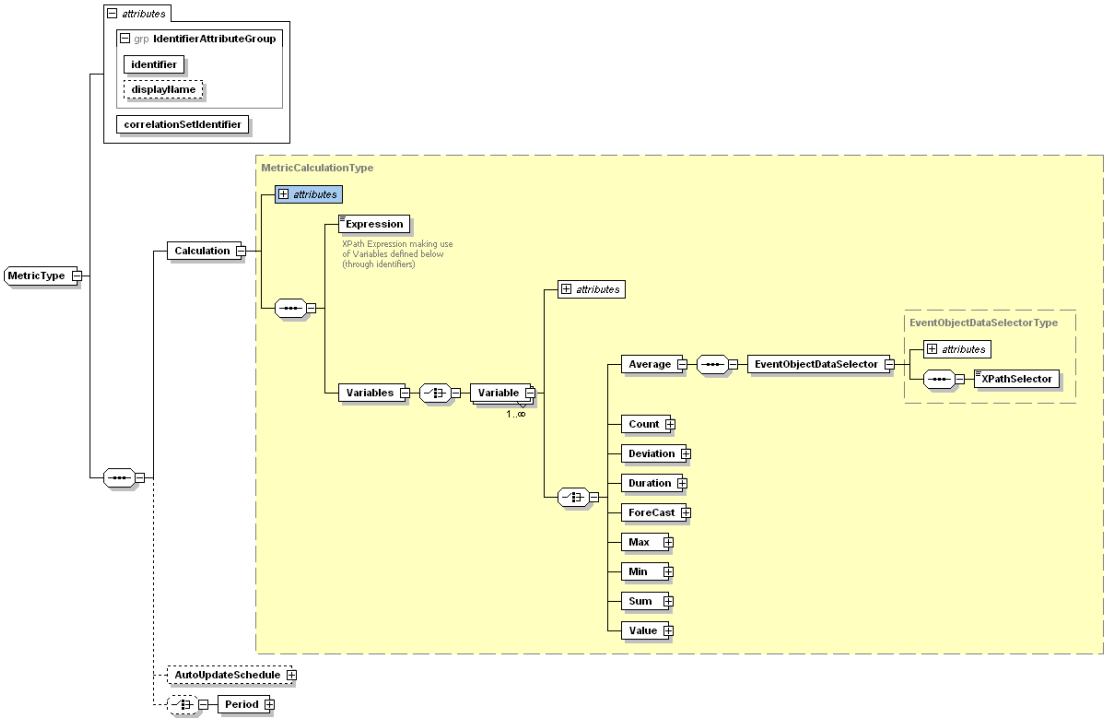

Figure 7.4: Declarative metric definition

Calculation consists of an *Expression* and multiple *Variables*. A *variable* is a value calculated from an event type in the correlation. We defined the following methods to calculate a variable:

Average calculates the average value of an event attribute of the specified event type.

Count counts how often the specified event type has occurred in the session

Deviation calculates deviation of an event attributes from a given value.

Duration calculates the timespan between two event types in the session

ForeCast calculates the forecast for an event attribute of the specified event type including the event attribute values already occurred in this session

Max, Min, Sum returns the max/min/sum value of an event attribute in this session.

Value Simply selects an event attribute from the specified event type.

Average, *Count*, *ForeCast*, *Max*, *Min* and *Sum* only make sense if an event type occurs more than once in a correlation instance. An example is the correlation *DataLoggerHistory* in Figure 8.1. All *DataLoggerRead* events of the same *DataloggerId* are collected in one session. To select the number of *DataLoggerRead* events in the session one can define a *Count* variable.

If the event type is only stored once per correlation instance, one would often use *Value* variables to select the value of a specific event attribute.

The *Expression* tag allows XPath Operators[1] to be executed on variables. The variables are replaced by their current values before the XPath Operations are executed. The result of the expression is returned as the current value of the metric. Because the metric calculation service will be executed for every session checkout, the metric is always up to date.

[1] http://www.w3schools.com/xpath/xpath_operators.asp

Listing 7.1: Simple Metric *ShipmentCosts*

```
<Metric correlationSetIdentifier="AllOrderInformation" displayName="CostOfAShipment"
 changeTracking="Auto" identifier="CostsShipment">
  <Description>Costs of a shipment</Description>
  <Calculation>
    <Expression>{var1}</Expression>
    <Variables>
      <Variable identifier="var1">
        <Value>
          <EventObjectDataSelector eventObjectTypeUri="eventobjecttype://TransportEnd">
            <XPathSelector>//Costs</XPathSelector>
          </EventObjectDataSelector>
        </Value>
      </Variable>
    </Variables>
  </Calculation>
</Metric>
```

Listing 7.1 shows a simple metric selecting only the *Costs* attribute from the *TransportEnd* event. The metric is stored with the identifier *CostsShipment* for every session of the correlationset *AllOrderInformation*.

Listing 7.2: Metric *Duration* with expression calculation

```
<Metric correlationSetIdentifier="AllOrderInformation" displayName="DemandReactionTime"
 changeTracking="Auto" identifier="DemandReactionTime">
  <Description>
   Starts right after Demand event gets in.
   The calculation is finished if a matching ShipmentCreated was received.
  </Description>
  <Calculation>
    <Expression>{var2} - {var1}</Expression>
    <Variables>
      <Variable identifier="var1">
        <Value>
          <EventObjectDataSelector eventObjectTypeUri="eventobjecttype://Demand">
            <XPathSelector>//DateTime</XPathSelector>
          </EventObjectDataSelector>
        </Value>
      </Variable>
      <Variable identifier="var2">
        <Value>
          <EventObjectDataSelector eventObjectTypeUri="eventobjecttype://ShipmentCreated">
            <XPathSelector>//DateTime</XPathSelector>
          </EventObjectDataSelector>
        </Value>
      </Variable>
    </Variables>
  </Calculation>
</Metric>
```

Listing 7.2 calculates the duration between the correlated *Demand* and the *ShipmentCreated* event. The *Expression* is used to calculate the difference between the two values. However, we could have used the *Duration* variable as well to calculate this metric.

Listing 7.3: Aggregated and filtered Metric *Max Costs*

```
    <Metric correlationSetIdentifier="AllOrderInformation" displayName="Max␣Cost␣Shipments␣to␣London"
     changeTracking="Auto" identifier="MaxCostsShipmentsLondon">
      <Description>Max costs of all shipment to London</Description>
      <Calculation aggregationMethod="Max">
```

```
5           <Expression>{var1}</Expression>
6           <Variables>
7             <Variable identifier="var1">
8               <Value>
9                 <EventObjectDataSelector xpathFilterExpression="//Location=London"
10                  eventObjectTypeUri="eventobjecttype://TransportEnd">
11                  <XPathSelector>//Costs</XPathSelector>
12                </EventObjectDataSelector>
13              </Value>
14            </Variable>
15          </Variables>
16        </Calculation>
17      </Metric>
```

Listing 7.3 shows two advanced concept:

aggregationMethod="Max" The attribute *aggregationMethod* calculates metrics not only on a specific correlation session, but also over all correlation sessions of a defined correlationset. Thus the aggregated metric stores, in this example, the maximum *Costs*-value in all *AllOrderInformation*-sessions. Available aggregations are **Average**, **Max**, **Min**, and **Sum**.

xpathFilterExpression="//Location=London" The attribute *xpathFilterExpression* restricts the metric calculation to the events that fulfill the given *xpathFilterExpression*. If the metric should only be calculated for specific correlation instances, a user can define a filter for every variable in their metric.

The example in Listing 7.3 returns the most expensive transport to London. This metric is not simply defined on a correlation session. With the combination of *aggregationMethod* and *xpathFilterExpression*, this metric is rather somewhere **between Level 2 and 3** as defined in Table 7.1. This metric could also be obtained by querying EventCloud for all transports to London, and then calculating the maximal costs in the resultset. Even we believe that EventCloud is the more flexible tool to calculate such queries, a user may need to know a specific metric all the time. In this case, it is better to predefine the metric and calculate it always during event processing rather than relying on the on-the-fly metric calculation of EventCloud.

7.5 Push Metrics: Alerts

Currently we use metrics to beautify and enrich the results returned by EventCloud. Metrics are stored to correlation instances supplying indicators about the

correlated events in these sessions. Metrics are a passive concept in EventCloud. They underlay the *pull-architecture* of EventCloud: only events and metrics a user explicitly searches for are displayed.

Until now we have focused on this pull-concept. However, with our event-driven approach, other usages of metrics can be found. In a *push-architecture*, a metric could be sent somewhere (to a user, another system, as email, etc.) every time it gets updated by an event. This way, real-time metrics are available for any other system or can be directly pushed to a user. [6] calls this principle *DBMS-Active, Human-Passive (DAHP)* model, whereas the pull-concept is called *Human-Active, DBMS-Passive (HADP)* model. We see the following advantages:

- metrics are not tied exclusively to EventCloud.
- other systems benefit from EventServer's event-driven, real-time, up to date metric capabilities.
- push information about the observed event scenario is available in real-time.

In a second step, we can utilize metrics to generate **alerts**. Metrics can be used to detect exceptions and unusual states within a scenario if a metric value is over a given threshold or the metrics are combined with Event-Condition-Action(ECA) rules. An alert signals to a user that something in the system is out of bounds. An alert is a push-concept and differs fundamentally from EventCloud which has its strength in event analysis. Nevertheless, currently a paradigm-shift in business intelligence from pull to push is taking place. Commercial tools are available under the buzzwords *Business Performance Management(BPM)* [10] and *Business Activity Monitoring (BAM)*. Instead of tons of reports generated by business intelligence tools on outdated data, these tools only send relevant information and/or exceptions to a contact person in real-time. As an example, I'd like to mention [19] which also offers a free community edition to see this concept on top of event streams in action.

With EventServer, we have the basis for alerts on metrics. When metrics are

defined as above, we can easily implement an event service like *"if MetricA is greater 1000 send email"*. But lots of event services, coded in Java, will implicitly code the "alarm-knowledge". This makes the system hard to configure and maintain. Again, it would be nicer to define alerts declaratively with XML on metrics. Another approach would be to use an existing rule engine that processes the events and metrics.

7.6 Conclusion

OLAP is the ideal for metric calculation and interaction as it offers all these possibilities with data drilling in the OLAP cube. As EventCloud uses a completely different data representation, by representing events and their correlations as documents, not all concepts of OLAP can be translated to our system. However, the concept of event correlation provides a powerful basis to calculate metrics for event analysis.

Many ideas on metric calculation came up during the development of EventCloud. Maybe the perfect solution is not found yet. However this chapter should give insight to the current status with metrics in event analysis. Further research in this area, like the implementation of a declarative metrics service, is desirable for learning more in this area.

8 Usage Scenarios for EventCloud

> *"The user is central to the success of an information retrieval system."*, Korfhage [16]

We saw the need for a new generation of analysis tools that directly acts on top of business events in Chapter 2. Until now, the technical requirements and their implementation have been discussed in detail, so this chapter focuses on usage scenarios which outline the added value EventCloud provided in complex event scenarios. The following use cases show the usage of EventCloud in the Medicare example, introduced in Chapter 2, from a *user point of view*.

The following stories should illustrate the idea of **event search and analyzing** and give some insight of the benefits and features the EventCloud system offers. We demonstrate the advantages of an event analyzing and search tool in multiple, different situations where they exceed approaches like OLAP, while in other situations it is still more accurate to apply OLAP.

8.1 Types of Search Queries

In [4], Battelle distinguishes between *recovery* and *discovery* search queries. Similar query categories can be found in EventCloud:

Recover This kind of search means queries where a user knows that some fact (a relevant result) is out there. They use a search tool to find the location of this information again. A simple example of this query type is the keyword-search in a web search engine to retrieve a specific URL. This kind of search is also referred to as *known item search* in Information Retrieval(IR) and represents the basic application of an IR system. The search for (correlated) events can

be executed in a similar way. If a user is interested in retrieving a specific event or a specific event correlation instance, they can execute a keyword search to "recover" this event.

Discover This kind of search means queries where a user thinks there is a relevant result out there. They explore the search space to find new or additional information, new connections and relationships that have not been obvious before. We know this kind of search from our own web experiences where we navigate through numerous webpages to find the answer to our question. The Internet supports this feature through hyperlinking. This type of search is the core essence for event analysis. **Discovery** can be seen as a retrieval and *browsing* task. Whereas a query returns the event that is the starting point for discovery, the browsing through events via links (represented by event correlations, see Section 6.1) allows to find additional information.

As EventCloud is a which makes the "web of events" searchable and navigable, both type of queries have to be supported. For *Recover*, the relevant events and correlationsets for the user query must be returned. This implies a relevance ranking algorithm for the resultset as discussed in 4.2. For *Discover*, the relevant events must be returned in the resultset, but additionally the user needs a facility to navigate through the events along meaningful connections. Connections in the EventCloud system are event correlations. Figure 6.2 shows how a user can navigate through events along these defined connections to explore and discover coherences.

8.2 Definition of the Medicare Example

For the following stories, I use a modified version of the Medicare example which was introduced by Szabolcs Rozsnyai in [28] and outlined in Chapter 2 to emphasize the need for event analysis tools. The example describes a TMS[1] CEP scenario for a pharmaceutical company. Every *Order* and *Transport* business process execution instance in the TMS system generates a number of events which are used as input events for the EventCloud system. Figure 8.1 shows six events described as:

[1]Transportation Management Systems

Demand The *Demand* event represents a demand occurring in one of Medicare's stocks.

ShipmentCreated If a demand event needs to be fulfilled from another location's stock, a *ShipmentCreated* event is created[2]. This event contains information about the transport that will be executed to satisfy the demand.

TransportStart The *TransportStart* event represents the actual start of a transport.

TransportEnd The *TransportEnd* event represents the actual end of a transport.

DataLoggerRead The *DataLoggerRead* event includes all measuring points of the datalogger sent along with a transport, so temperature violations during transports can be detected.

ShipmentReceived The *ShipmentReceived* event indicates that a transport has reached its destination. With this event, it is possible to infer whether the demand was satisfied successfully or not.

EventCloud relies heavily on the concept of event correlation as described in detail in Chapter 3. The correlation diagram in Figure 8.1 illustrates the six event types, their event attributes and how they could be correlated to each other in this scenario. Each colored line represents an independent correlationset. We see that an event can participate on multiple correlations. This makes sense as every correlation has a different meaning and could be interesting for event analysis. For example, the "AllOrderInformation" is interesting for event analysis as it allows to navigate through all process events of a specific "Demand causes Transport" episode. The correlations "Carrier performance" and "Datalogger History" are interesting for calculating performance metrics for a specific carrier or QoS metrics for a specific Datalogger[3].

For reasons of simplicity only the correlations "AllOrderInformation" (indirect correlation, include all event types), "shipmentInfo" and "transportInfo" are used in

[2]this decision may be made by a CEP application, so this event can be seen as "output event"

[3]As all transports a carrier X ever executes are stored in one correlation instance the corresponding correlation session can become very large

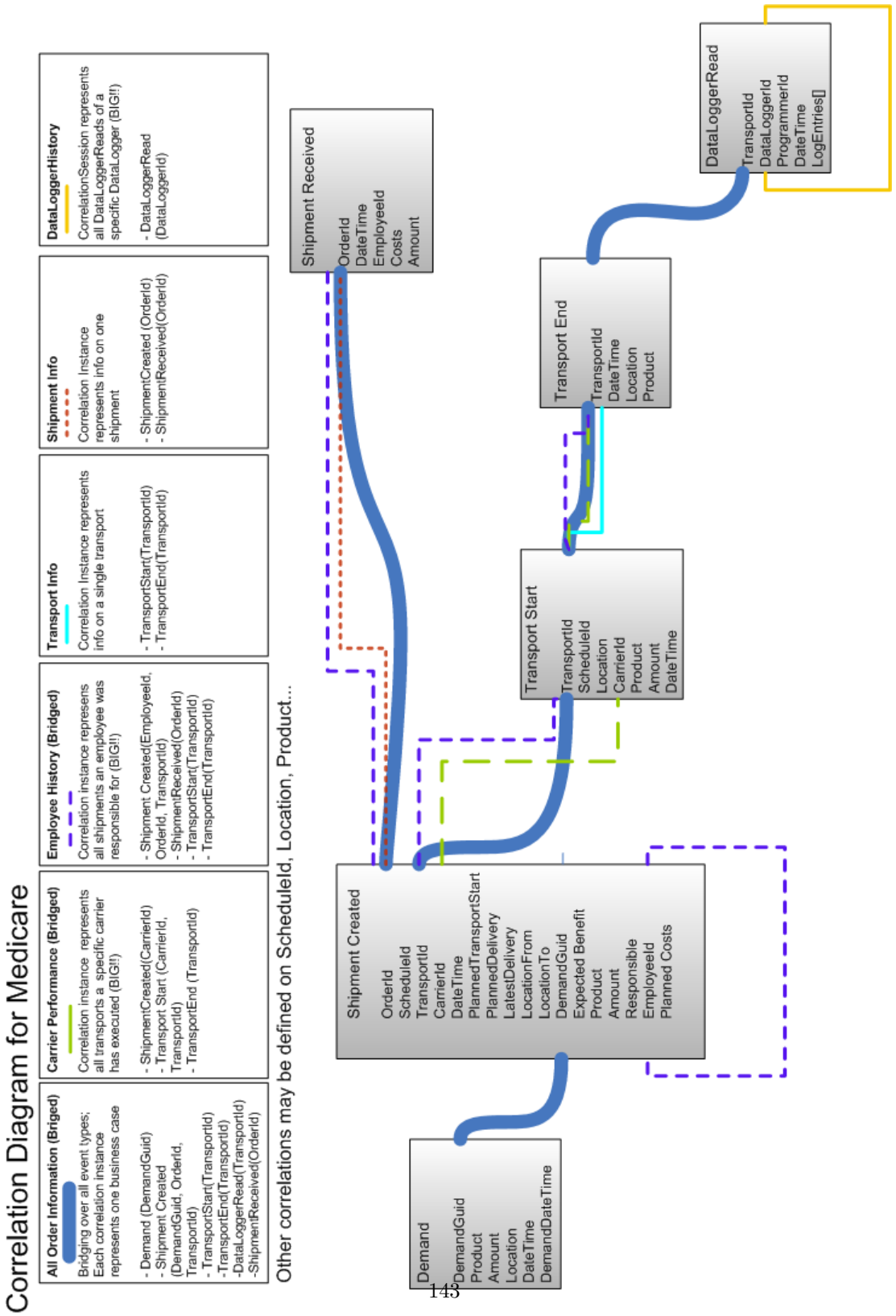

Figure 8.1: Correlation diagram of the modified Medicare example

the following examples.

Following are the metrics defined for the correlationset *"All Order Information"*. These metrics are calculated in the EventCloud system for every correlation instance, and are available for analysis:

Costs Order The costs of the shipment. Selected the attribute *Costs* from the *ShipmentReceived* event.

Duration Order Timespan of the whole order process in hours. Calculated from *ShipmentReceived.DateTime - Demand.DateTime*.

DELAY Delay of shipment in hours. Calculated from *ShipmentCreated.LatestDelivery - ShipmentReceived.DateTime*.

LOSS Amount of items lost during transport. Calculated from *ShipmentReceived.Amount - ShipmentCreated.Amount*.

COUNT TEMPVIOLATIONS Number of temperature violations logged by DataLoggerRead.

MAXTEMPVIOLATION Maximal temperature violation outside of the valid interval [-5, 10].

All these metrics are calculated on the three levels described in Chapter 7. Metrics have different meanings on each level:

Single correlation instance The metric is calculated during event processing for correlated events, e.g. for every transport. The value can be found in the EventCloud screenshots under the abbreviation ***CURR***.

Resultset The metric is calculated by EventCloud based on the metrics retrieved from the ***Single correlation instances*** *in the resultset* of a user query, such as the average costs for the five transport returned by the search query. The value can be found in the EventCloud screenshots under the abbreviation ***AVGRESULT***.

Defined correlationset The metric is calculated during event processing over all ***Single correlation instances*** *of a correlationset*, such as the average costs over all transports. The value can be found in the EventCloud screenshots under the abbreviation ***AVG***.

This preliminary information should make the following use cases understandable. This will illustrate how EventCloud works with the event types, correlations and metrics defined above to answer complex questions the **"event analysis"-way**.

8.3 Recovery Stories

8.3.1 Data Warehouse-like Usage

On September 1st, Peter Miller was instructed to generate a report about the performance of the carriers "MegaTrans", "Techtrans" and "TransTrasit" transporting the product "Tosalumn" between the locations Vienna and London in August. Whereas this would be a typical query in the DWH, this information is not yet available in the system. Since a decision who will execute future transports on this route must be made within the next hours, Peter can't wait for the next DWH update.

Peter queries EventCloud. He searches for the terms *"Tosalumn Vienna and London"* for August 2006 with the correlationset *"AllOrderInformation"* to get an overview over all transports made. In Figure 8.2, 21 transports with all their detailed process events are returned.

Peter is only interested in the overall metrics. He retrieves the following metrics from the resultset:

Transport OK: 86% 86% of all transports between Vienna and London in August had no problems. For all transports in the system the average is 89% so the current value is next to the average and does not indicate extraordinary problems.

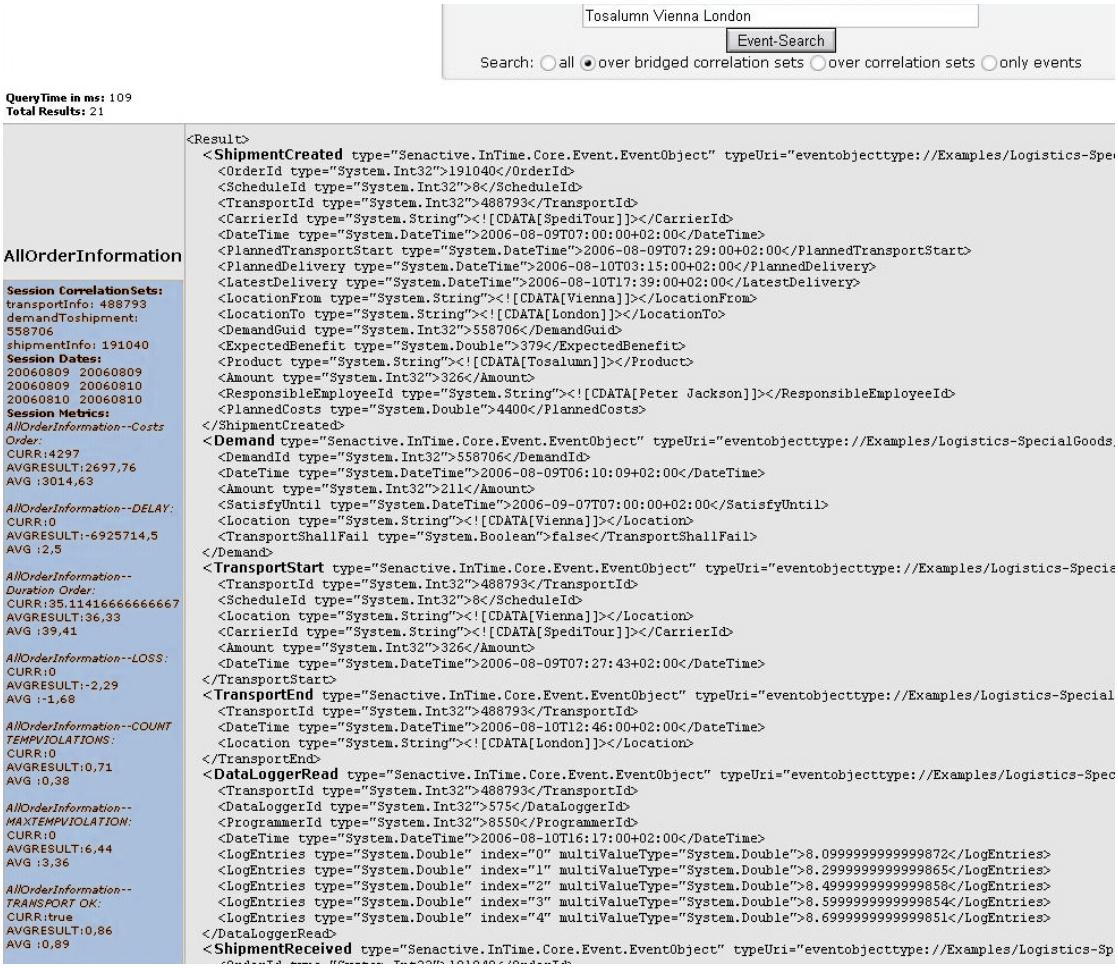

Figure 8.2: Queryresult in EventCloud: 21 transports returned

Costs Order 2697,75 The average order costs on this route are higher, however that's not very significant because Peter knows that other routes are longer and other products are more cumbersome than Tosulumn - so these transports are also more expensive.

To get individual metrics for the three carriers, he redefines his queries: he adds the carrier name to the query. Peter Miller starts with the evaluation of carrier "MegaTrans".

MegaTrans has taken 7 transports on this route in August. 71% were OK, however 2 transports failed. Shipment "714170" and "860543"(shown in Figure 8.4) both failed because of temperature violations during the transport. The average costs for MegaTrans transports were with a value of 2450 below the average on this route.

At Techtrans, one of its three transports failed on this route. Shipment 137624 failed for multiple reasons. Figure 8.4 shows the data that Peter can analyze for this shipment. It was delayed, 48 items out of 370 were lost, and multiple temperature violations occurred during the transport. As the metrics give a quick view that something went wrong, the events representing the process steps during the shipment give a detailed picture of how things went wrong.

With Transtrasit, all 8 transports went perfectly at an average cost of 2526. Peter is happy that at least one carrier seems to be a good candidate.

The three missing transports have been executed by carrier SpediTour. For some reason, the management had not mentioned this carrier. All transports went good, however the average price of 3445 is notably higher. Peter Miller creates a table out of the data he retrieved from EventCloud. With this table and a recommendation for the carrier Transtrasit, he attends to the management meeting and saves his company money and worries for future transports.

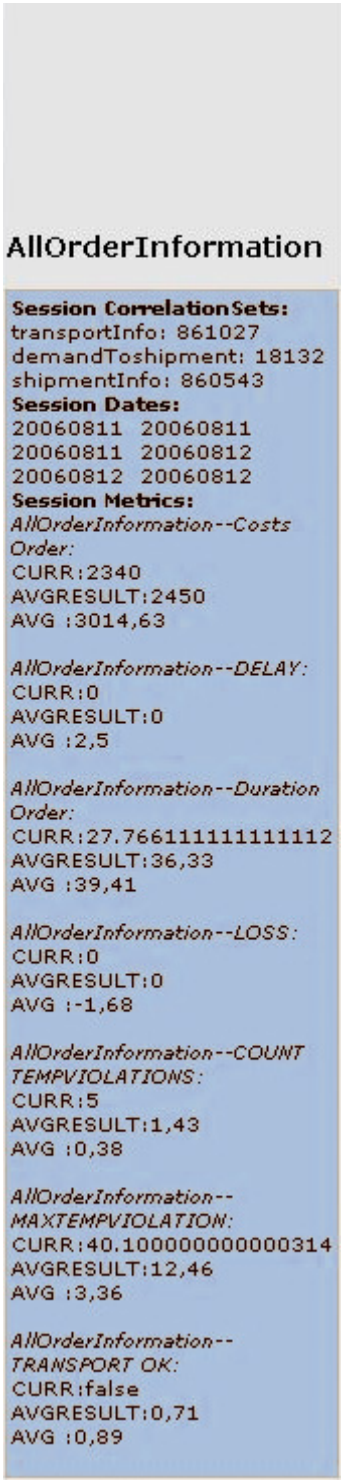

Figure 8.3: Metrics for the failed shipment 860543

Figure 8.4: Shipment 137624: Metrics show exceptions, whereas the detailed events can give you the exact information (see here for loss and temperature violation [temp should be between -5 and 10])

	Vie-Lon	All	MegaTrans	Techtrans	Transtrasit	SpediTour
Transports Total	21	-	7	3	8	3
Transports OK	86%	89%	71%	67%	100%	100%
AVG Costs	2698	-	2450	2986	2526	3445

Table 8.1: Metrics for route Vienna - London August 2006

How EventCloud was used

In the story above, we see a simple usage of EventCloud. Search is restricted to a specific correlationset and timespan. The correlationset *"AllOrderInformation"* stores all the metrics the user is interested in. By refining the query (with the carrier name), the user can reduce the resultset and receive the metrics on the resultset-level he is interested in. The detailed event view allows the user to see exactly why transport exceptions have been detected during the process execution. Not only is aggregated information available within EventCloud, but the detailed, fine-grained, process-oriented information in the events can be retrieved.

In future versions of EventCloud it should also be possible for a user to create reports within EventCloud. Currently, EventCloud simply displays the figures, but as we see for Table 8.1 a user needs a second tool like Excel to collect, edit and prepare data for a presentation.

8.3.2 Realtime usage

Sandra Goodman was notified that the transport for OrderId 306717 has been delayed. To get some additional background on this transport she wants to contact the responsible employee for this shipment. She queries EventCloud for the term *"306717"*. As a result in Rank3 she gets a single resultset of the correlationset *"AllOrderInformation"* as shown in Figure 8.5. She takes a look at the correlated events, showing her the details of the transport. By now the *Demand*, the *ShipmentCreated* and the *TransportStart* event have been processed. No metrics could have been calculated since they depend on the *ShipmentReceived* and *DataloggerRead* events. Sandra infers from the current events that the transport is somewhere stuck between T*ransportStart.Location* **Vienna** and *ShipmentCreated.ToLocation* **Munich**. Looking at the details, she realizes that *TransportStart.DateTime* was delayed to *ShipmentCreated.PlannedTransportStart*.

From the *ShipmentCreated* event Sandra gets the responsible contact person, Peter Jackson. Calling Peter minutes after the delay notification gives Sandra a possi-

Figure 8.5: Current status of Shipment 306717. The shipment's correlation instance is available immediately in EventCloud

bility to replan the transport. Peter tells her that a new transport can be shipped by plane within one hour. Sandra compares this offer with the planned costs from the *ShipmentCreated* event and agrees to the new plan.

How EventCloud was used

As EventServer is indexing the events processed in near real-time, data is immediately available in EventCloud. Event though the events *TransportEnd* and *ShipmentReceived* were not yet received, the current state of the transport is always available in the search index, although not all metrics can be calculated. The detailed event view allows a user to dig into the correlated events and extract the specific information they are interested in.

8.3.3 More Recovery Stories

Steve Moore needs to know why the transport of "Tosalumn" on 7. August 2006 to Vienna has failed. He queries EventCloud for the terms *"Tosalumn 07.08.2006 Vienna FAILED"*. As a result he gets the *Transport 20475* from Paris to Vienna. The transport actually failed because of a temperature violation. The maximal

temperature violation of 48 degrees occurred during a timespan of 4 hours.

EventCloud allows inference from multiple search terms to a specific case. The search and ranking capabilities of EventCloud help to find (correlated) events even if one does not have exact identifiers for them.

The carrier "Mechatronic" claims he has a performance of 99 OK transports with the last 100 Transports. Kelly Taylor is asked to validate this statement. She queries "Mechatronic" and cuts the searchresult to the last 100 transports. The metric displays 92 transports OK, 8 transports NOT OK. Kelly adds "NOT OK" to her query and receives exactly 8 transports with detailed information what went wrong on each of these transports.

EventCloud sets on top of a boolean query language[4] and allows search terms to be combined through logical operators.

8.4 Discover Stories

8.4.1 Search and Navigation

Kathy Jackson has to review Transport 20475 which failed because of temperature violations. She queries for "Transport 20475" to get a view on the transport. EventCloud returns the correlation *AllOrderInformation* that represents this business case. The Datalogger 777 has detected three temperature violations of 48 degree Celsius. Kathy clicks at the Datalogger's identifier to retrieve the history of this datalogger (she navigates to the correlation set *DataLoggerHistory* (Figure 8.1)). The history metric shows that the datalogger has detected three temperature violations in the last transports. Kathy is stunned! She takes a closer look at the other two transports, 20111 and 20214, and notices that the datalogger always logged a violation of 48 degree Celsius. Obviously the datalogger is defect, so Kathy contacts the datalogger department to inspect Datalogger 777!

[4] see http://lucene.apache.org/java/docs/queryparsersyntax.html

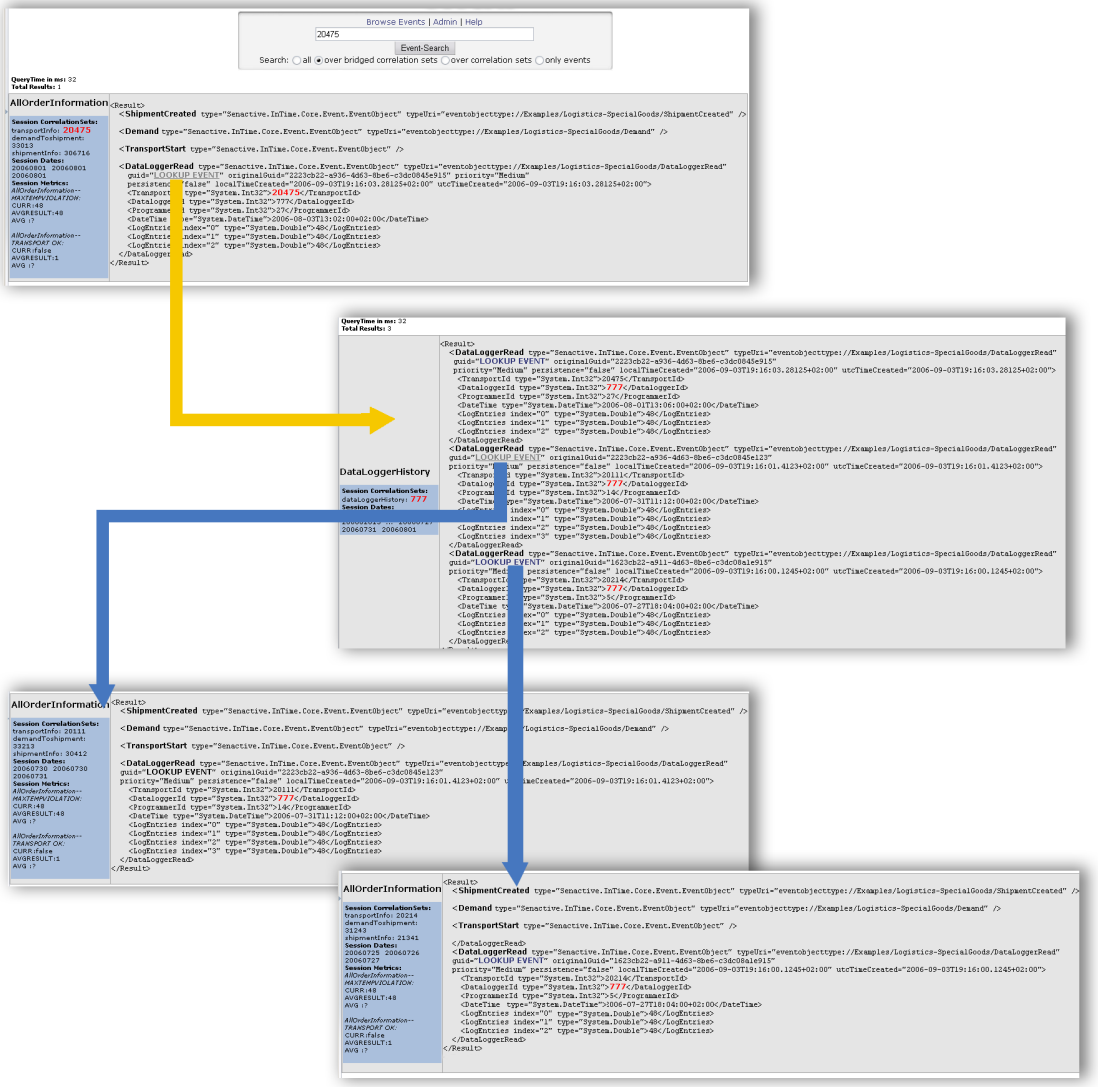

Figure 8.6: Discovery example: finding an exceptional situation by navigating between correlations.

Figure 8.6 shows the analysis steps with EventCloud. First the user queries for a term, like the TransportId 20475. The resultset returns the matching correlation instance *AllOrderInformation* for the transport. The user now has a detailed view on the transport. To navigate from an event of the correlation to other correlations, the user has to click the link "LOOKUP EVENT". This causes a new query, returning all correlation instances this specific event participates. In the figure, one of these correlations is the correlation *DataLoggerHistory*. The *DataLoggerHistory* correlation collects all events of a specific DataLogger, in this case of the Datalogger 777 (see Figure 8.1). From the DataLoggerHistory correlation it is again possible to navigate to other correlations, like to the two transports 20111 and 20214.

How EventCloud was used

Event correlation is used to navigate from a single business case correlation "Transport 20475" via a specific *DataloggerRead* event to other correlated events. With the help of correlation the user walks through the cloud of events to find information that is not obvious from a single event or even a single case. As the detailed event data is available in EventCloud the user can detect eye-catching patterns like the defect of the datalogger.

8.4.2 More Discovery Stories

During the last quarter, 25 transports with loss have been detected on shipments to London. Paul Smith analyzes these cases. He queries EventCloud for "London Loss" selecting the last quarter as time range. 25 transports are listed in the result set. On average, three items have been lost on these transports. After analyzing the first three transports, he realizes that the carrier is always "Transit". He navigates to the list of all transports "Transit" made during the last quarter. As "Transit" is one of the most common carriers, 229 transports are found where 29 were declared as failures. Adding "London" to the current view he gets 35 results, including the 25 failed transports. Paul realizes that this is a very bad number and makes further analysis. All of the transports are coming from Rome! He

immediately contacts the carrier to tell him about these numbers. When "Transit" made an internal control of this incidence, an employee has admitted that he has stolen 105 items during the last quarter.

EventCloud allows an analyst to find new information and exceptional states that are not simply detected otherwise. With the help of event search and navigation and through multiple analysis steps the user finds a complex reason for a simple observation

9 Conclusion

9.1 Analysis tool for CEP

This thesis discusses the need for tools to analyze and explore business events as a consequence of the emerging technologies Complex Event Processing (CEP) and Business Activity Monitoring(BAM). As todays business is event-driven, CEP gains more and more attention because of its capability to process these events in real-time by correlating simple events into higher-level, complex events to identify opportunities, potential problems and exceptions. This allows enterprises to evolve from today's reactive behavior to a proactive behavior by sensing current business situations and actively responding immediately to them. CEP, therefore, needs to offer a toolset of technologies as described in [22].

CEP solutions have their strength in running 24*7 to process large amounts of events to proactively handle current business situations. They implement non-trivial domain-knowledge in the form of continuous queries, rules or user functions which are applied on the stream of events received. That means CEP applications run autonomously without user interaction after they are set up for a specific use case once. CEP automatically generates alerts, adapts the business process and/or notifies responsible employees.

It is not sufficient to have a CEP solution but to do not have any insight what is happening inside, or better, why it is happening. From an enterprise perspective, a CEP is a blackbox that may perform well, or not so well, measurable in numbers on business performance. However, the person in charge needs to state a reason for the decisions made by their system, wants to investigate in specific situations

or wants to find new opportunities.

There is a need to have event analysis tools that support CEP applications and their users by allowing analysis of the events put into and received from the system. This tool must provide features which help to understand the business process with its input events, the decision-finding and the results delivered by the CEP application. The tool enables analyzing the processed events offline, detached from the real-time processing done by the CEP, to allow an analyst (a domain expert) to draw conclusions, adapt the domain-knowledge of the CEP application, and enhance the business process.

EventCloud is proposed as such a generic *Analysis Tool* for event processing scenarios. The EventCloud frontend enables the user to navigate through events, pick up single events and display their content, discover chains of events and how they are correlated, and to recognize patterns in the cloud of events. Metrics can be defined to provide insight at a higher level and to measure business performance. It visualizes business and process developments at the fine-grained level of events.

9.2 Characteristics of Event Analysis

Chapter 2 described why today's technologies like OLAP are not sufficient to create an event analysis tool. By mapping events into a relational data representation like a star-schema, information is lost that is included in the original events. Using a relational schema has drawbacks: as CEP application processes events, mapping events from CEP to a database is expensive, needs to be done for every CEP scenario separately and still does not include all information. Instead of building a data oriented schema used by OLAP, the representation should be "event-oriented" to reflect the business process. It is natural to produce a data schema which directly represent the events, their occurrence in time and their correlations to each other.

EventCloud introduces a data schema based on top of events and their corre-

lations. Chapter 4 describes the concepts of this document oriented approach and it's relation to Information Retrieval in detail. Rank1, Rank2 and Rank3 were introduced to represent different levels of event searching to apply relevancy ranking and to allow non-trivial queries. The data schema was implemented with Apache Lucene which creates a full-text index over all processed events. EventCloud allows searching and navigating in (correlated) events to answer complex queries in a large set of events, partly comparable to OLAP analysis and partly offering new ways of analyzing. The Medicare use cases example in Chapter 8 shows the opportunities EventCloud offers.

9.3 EventServer and EventCloud

Learning from the previous implementation of EventCloud [28], the architecture of the EventCloud system has completely changed and improved. It exceeds the original approach in functionality, design and performance as many things have been learned since then.

EventCloud now itself has the characteristics of a CEP system as it processes the events in real-time to produce the event data representation. To make a clear distinction between event processing and the EventCloud user frontend, I introduced the EventServer. The EventServer is a platform which provides functionality for implementing CEP applications. EventServer offers essential system services like event correlation, a multi-threaded architecture for event processing, adapters to other systems via JMS, and the possibility to implement user functionality for any event processing task through the concept of event services.

For this reason, the EventCloud backend is simply an application deployed in the EventServer. The EventCloud backend implements specific event services that process events in real-time to create the data basis for event analysis. On top of this data representation, the EventCloud user frontend was implemented to offer a rich frontend allowing a user-friendly search, analysis and exploration of events.

Functionality of the EventCloud system was improved by allowing the search in indirectly correlated events and by adding the concept of metric calculation on top of correlated events. By extending the original event correlation concept with indirect correlation to increase the query power of EventCloud, I had to solve numerous new difficulties which are addressed in this thesis. Metrics on top of correlated events are a new approach in EventCloud which became possible due to the architecture of the EventServer. Metrics enrich the search result of EventCloud and brings it closer to the functionality OLAP offers.

9.4 Future Research

With the EventServer, the infrastructure for arbitrary ideas and applications in the area of event processing has been created. As it implements the event correlation approach presented in Chapter 3 everybody can implement their own event services making use of event correlation. We came up with many ideas for event processing scenarios besides event analysis with EventCloud. One is the observation of software projects by collecting and correlating events from the different IT-systems used in software development like version control, bug tracking, and mailing lists. Another idea regards the detection and visualization of fraud at telephone companies. Even others go in the direction of event stream processing and event mining. By using the EventServer it is not necessary to start from zero with each new project.

EventCloud and the concepts of event analysis have greatly evolved through my endeavors. Future research needs to concentrate on improving the frontend by implementing, in addition to the current text-view, ideas from information visualization. Information visualization can help a user to easier navigate events and visually detect patterns within the large amount of events. SENACTIVE currently implements a commercial event analysis tool with a focus on visualizing events.

Further research needs to be done in the area of event mining and pattern detection. While EventCloud helps finding unknown information and, in combination

with accurate information visualization, suggests hypotheses to the users, we need to investigate how known methods from data mining and statistics can be applied on events and their correlations.

9.5 Conclusion

In this thesis the EventCloud system is applied to a complex event processing scenario for the first time. The Medicare use cases example shows the different nature of event analysis as compared to todays data-oriented approaches like OLAP. As the underlying data representation differ completely, adequate concepts have been developed and presented to allow search, analysis and exploration of correlated business events.

It shows that event analyzing is an essential part of the CEP toolkit. It is the missing link between real-time event processing, done by CEP applications, and analysts and domain experts which need to understand, monitor and enhance these CEP solutions.

10 Appendix

10.1 Research Topics

Append() Functionality As described in Section 6.3.1 the performance of the current implementation suffers from the disability to append events to existing documents, as Apache Lucene does not offer an accurate functionality for this requirement. As correlations grow with every event, this is a major limitation. Correlations(Documents) needs to be re-indexed with every new event. Further research must be done if and how an append-functionality can be realized.

Extending Query Power Currently the documents (events/correlations) are retrieved by a user query if all query terms can be found somewhere in the document. That is not sufficient for every situations, for example if a user wants to search for a specific event attribute in a specific event type. Lucene offers ways to flatten the event XMLs before indexing them. Other full-text indexes like MS SQL Server 2005 offer XQuery-statements to search for more than just terms. Additionally not only events, but also metrics should be searchable for operations like EQUALS, LESS, MORE. This way EventCloud could retrieve all correlations where the costs are e.g. larger than a given value.

Managing distributed indexes In Section 6.4.4 multiple approaches were discussed to create distributed indexes to handle large amounts of event data. These approaches need to be refined and implemented.

Frontend Section 5.4 has shown in which direction event analysis has developed with SENACTIVE EventAnalyzer. EventCloud's web-frontend should evolve

and generate new ideas how users can be supported in their event analysis tasks. Visualizing events and their correlation has become a major topic.

Metric calculation In Section 7.4 a reusable event service for generic metric calculation with XML declarations was described. This is an important service for the EventServer as it is reusable in many situations, from pull-architectures like EventCloud to push-architectures like metric alerts described in Section 7.5.

IFS EventServer The reimplementation of the EventServer has already started. Reusing the concepts of this thesis (event correlation, event services) and extending them by using an ESB, powerful event routing possibilities, having numerous adapters and a scalable architecture will provide a mature event processing platform. An integration of the ESP Esper would be exciting, as well as a graphical frontend to simplify the management of the server and the event processing applications.

CorrelationService @ EventServer; Pseudocode

```
checkoutSession (correlationName, List<String> correlationSets>
    Sort correlationSets to avoid deadlocks
    prepareCheckoutExecution (correlationSets)
        for each correlationValue in correlationSets
            enter monitor for correlationvalue
            → nobody else is allowed to access with this correlationvalue (other requests with the same correlationValue are blocked)
            lookupSessionGUID(correlationValue) → lookup the sessionGUID for given correlationValue
            if null → no session available for this correlationValue; store correlationValue in temporary list
            NoSession4CorrelationValue
            else → (if I havn't already checked out the sessionGUID with another correlationValue before(optimization)):
            Session = request2TakeSession (sessionGUID,correlationValue)
                enterMonitor für sessionGUID → synchronize session access
                getSession from database
        //By now I have a monitor for all my correlationValue. This avoids race conditions
        //All sessions that exists for the given correlationValues are monitored and checked out
        //I know all correlationValues that do not point to a session by now
    //If no correlationValue points to a session → create a new Session
    createNewSession (correlationName, correlationSets)
        create a new session
        enter Monitor for session → sync session access
        persist session
        addCorrelationSets2Session (correlationSets,session)
            store correlationValues in Session
            (the session must know which correlationValues point to it [for EventGoogle, SessionMerge])
            For all correlationValues in correlationSets
                addOrUpdateSessionLookupCorrelationData(session, correlationValue)
                    update lookup of correlationValue to sessionGUID
                    update lookup of correlationValue to sessionGUID in database
        return new Session

    else merge (simple and complex) :

    mergeSessions(neededSessionInfo.getMySessions(), neededSessionInfo.getNoSession4CorrelationValue());
        pick first session in list as Merge Winner
        for each other session checked out → Complex Merge if sessions in loop
            addCorrelationSets2Session(helperSession.getCorrelationSetsAsString(),mergeSession)
            → update lookup of correlationValue to sessionGUID
            simpleParametersMerge(mergeSession, helperSession)
            → add content of session in the Merge Winner session
            mergeSession.addMergedCorrelations
            → add the sessionGUID to a list in the MergeWinner session (to give indexing the needed info which merge happend)
            sessionQueueMonitor.mergeQueues (helperSession, mergeSession)
            → merge all threads waiting for this session into the Merge Winner monitor queue
            Delete this session
        addCorrelationSets2Session(noSession4CorrelationValue, mergeSession)
        → Simple Merge: for each correlationValue in neededSessionInfo.getNoSession4CorrelationValue()
          add these correlationValues to the Merge Winner and update their lookup to Merge Winner's sessionGUID.
        Return the single, (merged) session Merge Winner.
    Return session Merge Winner from CorrelationService to caller
```

Figure 10.1: Pseudocode for EventServer's correlation service implementation

Bibliography

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. *Proceedings of the CIDR'05*, 2005.

[2] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 2001.

[3] Baeza-Yates and Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, 1999.

[4] J. Battelle. *The Search: How Google and Its Rivals Rewrote the Rules of Business and Transformed Our Culture*. Penguin Books, 2005.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Proceedings of the seventh international conference on World Wide Web 7*, 1998.

[6] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. *Proceedings of the VLDB'02*, 2002.

[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. *Proceedings of CIDR Conference'05*, 2003.

[8] O. S. Compass. Compass. http://www.opensymphony.com/compass/, 10 2006.

[9] S. GmbH. Senactive intime. http://www.senactive.com, 03 2006.

[10] M. Golfarelli, S. Rizzi, and I. Cella. Beyond data warehousing: What's next in business intelligence? *Proceedings of the 7th ACM international workshop on Data warehousing and OLAP*, 2004.

[11] O. Gospodnetic and E. Hatcher. *Lucene in Action*. Manning, 2005.

[12] T. S. Group. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin, 26(1)*, 2003.

[13] R. Hackathorn. Current practices in active data warehousing. *DMReview*, 2002.

[14] http://esper.codehaus.org/. Esper. http://esper.codehaus.org/, 10 2006.

[15] A. S. Josef Schiefer. Management and controlling of time-sensitive business processes with sense and respond. *International Conference on Computational Intelligence for Modelling Control and Automation (CIMCA)*, 2005.

[16] R. R. Korfhage. *Information Storage and Retrieval*. Wiley, 1997.

[17] D. Lewandowski. *Web Information Retrieval*. Deutsche Gesellschaft für Informationswissenschaft und Informationspraxis, 2005.

[18] H. Lieberman. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26:419–429, 1983.

[19] S. S. Ltd. Seewhy real time business intelligence. http://seewhy.com, 2006.

[20] A. F. Lucene. Apache lucene. http://lucene.apache.org/, 10 2006.

[21] A. F. Lucene. Lucene performance benchmarks. http://lucene.apache.org/java/docs/benchmarks.html, 10 2006.

[22] D. Luckham. *The Power Of Events*. Addison Wesley, 2002.

[23] D. C. Luckham. Rapide: A language and toolset for simulation of distributed systems by partial orderings of events. *DIMACS Partial Order Methods Workshop IV*, 1996.

[24] C. Nicholls. In search of insight. http://www.seewhy.com, 10 2006.

[25] O. D. Norman W. Paton. Active database systems. *ACM Computing Surveys (CSUR) Volume 31*, 1999.

[26] L. Perrochon, S. Kasriel, and D. C. Luckham. Managing event processing networks. (CSL-TR-99-788), 1999.

[27] A. Rappoport and S. T. Consulting. http://www.searchtools.com/. http://www.searchtools.com/, 09 2006.

[28] S. Rozsnyai. Efficient indexing and searching in correlated business event streams. Master's thesis, Technical University Vienna, 2006.

[29] J. Schiefer and C. McGregor. Correlating events for monitoring business events. *ICEIS*, 2004.

[30] R. W. Schulte. Introducing the zero-latency enterprise. Gartner Research, 6 1998.

[31] R. W. Schulte. The growing role of events in enterprise applications. Gartner Research, 7 2003.

[32] I. StreamBase Systems. Streambase. http://www.streambase.com/, 10 2006.

[33] I. StreamBase Systems. Streamsql. http://blogs.streamsql.org/, 10 2006.

[34] A. Umar. It infrastructure to enable next generation enterprises. *Information Systems Frontiers*, 7, 2005.

[35] R. Vaarandi. Sec: a lightweight event correlation tool. *Proceedings of the 2002 IEEE Workshop on IP Operations and Management*, 2002.

[36] I. A. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, 1999.

[37] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D.Ohsie. High speed and robust event correlation. *Communications Magazine, IEEE*, 34, 1996.