EVENT PROCESSING ENGINE FOR STREAM DATABASES (DISANG ENGINE)

A B. Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Abhimanu Kumar
(03010101)

under the guidance of

Dr. G. Sajith

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**EVENT PROCESSING ENGINE FOR STREAM DATABASES (DISANG ENGINE)**" is a bonafide work of **Abhimanu Kumar (Roll No. 03010101)**, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Dr. G. Sajith**

Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology Guwahati, Assam.

May, 2007
Guwahati.

Acknowledgements

I would like to express my sincere thanks to Dr. G. Sajith, Associate Professor, Department of Computer Science and Engineering, IIT Guwahati, for his invaluable guidance during the course of project. I am highly indebted to him for constantly encouraging me by giving his critics on my work. I am also thankful to various Complex Event Processing and Event Stream Processing Software communities whose discussions and web pages provided me invaluable help in making my own Event Processing Engine, Disang.

Contents

List of Figures vi

List of Tables vii

1 Introduction 1
- 1.1 Disang (Our Event Processing Engine) . . . 2
- 1.2 Features of the Disang Engine . . . 2
- 1.3 Organization of The Report . . . 4

2 Traditional Database and Stream Database 5

3 Event Stream Processing and Stream Database 7
- 3.1 Queries over Data Stream . . . 9
  - 3.1.1 Unbounded Memory Requirements . . . 9
  - 3.1.2 Approximate Query Answering . . . 9
  - 3.1.3 Sliding Windows . . . 10
  - 3.1.4 Batch Processing, Sampling, and Synopses . . . 11
  - 3.1.5 Blocking Operators . . . 11
  - 3.1.6 Queries Referencing Past Data . . . 12
- 3.2 Query Language for a DSMS . . . 12
- 3.3 Timestamps in Streams . . . 13

4 Our Aim, the System and the Problem 14
- 4.1 Our Aim . . . 14
- 4.2 The System . . . 15
- 4.3 The Problem . . . 18

5 Previous Works 19
- 5.1 STREAM (Stanford Stream Database Manager) . . . 19
  - 5.1.1 Abstract Semantics . . . 20
  - 5.1.2 Relation-to-Relation Operators in CQL . . . 21
  - 5.1.3 Relation-to-Stream Operators in CQL . . . 21
  - 5.1.4 Query Plans and Execution . . . 22
  - 5.1.5 Operator . . . 22
  - 5.1.6 Queue . . . 22

5.1.7 Synopses 22
5.1.8 Query Plan Execution 22
5.2 Esper 23
5.3 A few other projects 23

6 The Disang Engine 24
6.1 Configuration Component 24
6.2 Scheduling Component 25
6.3 Event Component 26
6.4 EQL (Event Query Language) Component 27
6.5 Parser Component 28
6.6 Miscellaneous other Components 30
6.6.1 AutoImport Services 30
6.6.2 Database Services 30
6.6.3 Timer Services 30
6.6.4 Emit Services 30
6.6.5 Dispatch Services 30
6.6.6 View Services 30
6.6.7 Stream Services 30
6.6.8 Read-Write Lock Services 31
6.6.9 Runtime coordination Services 31
6.6.10 Administrative Services 31
6.7 Initialization and the Working of the Engine 31

7 Running Example of the engine (Solving the Self Service Terminal Problem) 35
7.1 Events as JavaBeans 35
7.2 Introduction to EQL and Patterns 38
7.3 Registering Statements and Listeners 38
7.4 Detecting the Absence of Status Events 39
7.5 Activity Summary Data 40

8 Conclusion and Future Work 42

References 43

List of Figures

3.1 Stream Database structure 8

4.1 Waiting in line for toll booths is starting to become a thing of the past as variable tolling begins to emerge and better technology is developed. 15
4.2 Flight Management System 16
4.3 Event cloud in a terminal managing system 17

5.1 Stream Database structure 20
5.2 Data types and operator classes in abstract semantics. 21

6.1 scheduling of different Views 25

7.1 Self service terminal set 36
7.2 The Status Event Stream (The output first clause can suppress output events) 40

List of Tables

6.1 Some feasible Events 26

7.1 Events of a self-service terminal 36

Chapter 1

Introduction

In many recent applications, data takes the form of continuous, unbounded data streams, rather than finite stored data sets. Examples of data streams include stock ticks in financial applications, performance measurements in network monitoring and traffic management, log records or click-streams in Web tracking and personalization, data feeds from sensor applications, network packets and messages in firewall-based security, call detail records in telecommunications, and so on. In all of the applications cited above, it is not feasible to simply load the arriving data into a traditional database management system (DBMS) and operate on it there. Traditional DBMS's are not designed for rapid and continuous loading of individual data items, and they do not directly support the continuous queries [DTO92] that are typical of data stream applications. Furthermore, it is recognized that both approximation and adaptivity are key ingredients in executing queries and performing other processing (e.g., data analysis and mining) over rapid data streams, while traditional DBMS's focus largely on the opposite goal of precise answers computed by stable query plans. Stored data set is appropriate when significant portions of the data are queried again and again, and updates are small and/or relatively infrequent. In contrast, a data stream is appropriate when the data is changing constantly (often exclusively through insertions of new elements), and it is either unnecessary or impractical to operate on large portions of the data multiple times. Today's database systems are ill-equipped to perform any kind of special storage management or query processing for data streams, heavily stream-oriented applications tend to use a DBMS largely as an offline storage system, or not at all. Here we consider fundamental models and issues in developing a general-purpose Data Stream Management System (DSMS). Alongside we attempt to provide a general overview of the area, along with its related and current work. We discuss also plethora of past research in areas related to data streams: active databases, continuous queries, filtering systems, view management, sequence databases, etc. So we have made an engine which processes stream events and outputs the required results.

1.1 Disang (Our Event Processing Engine)

This is our basic Stream Database Processing Engine written in Java. We have named it "Disang". Disang helps in quick development of applications that process great amounts of incoming messages or events. Disang filters and analyzes events in many ways, and reports back situations of interest in real-time. The Disang engine is created to address the needs of applications that analyze and react to events. Some common applications are:

- Business process management and automation (process monitoring, reporting exceptions, operational intelligence).
- Finance (algorithmic trading, fraud detection, risk management).
- Network and application monitoring (intrusion detection, SLA monitoring).
- Sensor network applications (RFID reading, scheduling and control of fabrication lines, air traffic).

Complex Event Processing, or CEP, is technology to process events and discover complex patterns among multiple streams of event data. ESP stands for Event Stream Processing and deals with the task of processing multiple streams of event data with the goal of identifying the meaningful events within those streams, and deriving meaningful information from them.

1.2 Features of the Disang Engine

Our Engine has following features:

- Event Stream Processing
  1. Time-based, interval-based, length-based and sorted windows
  2. Grouping, aggregation, sorting, filtering and merging of event streams
  3. Tailored SQL-like query language using insert into, select, from, where, group-by, having and order-by clauses
  4. Inner-joins and outer joins (left, right, full) of an unlimited number of streams or windows
  5. Subqueries including exists and in
  6. Output rate limiting and stabilizing
- Event Pattern Matching
  1. Logical and temporal event correlation
  2. Crontab-like timer 'at' operator
  3. Lifecyle of pattern can be controlled by timer and via operators
  4. Pattern-matched events provided to listeners

- Event Representations
    1. Supports event-type inheritance and polymorphism as provided by the Java language
    2. Events can only be plain Java objects(In future this can be extended to XML document object model (DOM) and java.util.Map including nested objects)
    3. Event properties can be simple, indexed, mapped or nested - allows querying of deep Java object graphs (and may be XML structures in future)
- Event Pattern Matching
    1. CSV input adapter reads comma-separated value formats, and can simulate multiple event streams with timed playback
    2. JMS input and output adapter based on Spring JMS templates
- Relational database access via SQL-query joins with event streams; LRU and expiry-time query result caches
- Supports both listener (push) and consumer (pull) model
- Supports externally-supplied time as well as Java system time

What these applications have in common is the requirement to process events (or messages) in real-time or near real-time. This is sometimes referred to as complex event processing (CEP) and event stream analysis.

Key considerations for these types of applications are the complexity of the logic required, throughput and latency.

- Complex computations - applications that detect patterns among events (event correlation), filter events, aggregate time or length windows of events, join event streams, trigger based on absence of events etc.
- High throughput - applications that process large volumes of messages (between 1,000 to 100k messages per second)
- Low latency - applications that react in real-time to conditions that occur (from a few milliseconds to a few seconds

We will cover the working, code, structure and other properties of our Engine in detail in the following chapters. The above discussion was just an outline of what is all we are going to talk about later.

1.3 Organization of The Report

In a brief outline of what we are going to cover further in our report is as follows. We begin by contrasting between traditional database and Stream database in chapter 2 then we discuss in detail about the Event Stream Processing and Stream Database and what is it all about along with presenting some concrete examples to ground our discussion in Chapter 3. Then in Chapter 4 we discuss our aim and desired result also providing a fine outline of the system we have to work with and the problem we want to solve in that system. Then in Chapter 5 we discuss any previous work carried out in this direction. We review recent projects geared specifically towards data stream processing further moving more deeply into the area of query processing, uncovering a number of important issues. After that we outline some details of a query language and architecture for a Complex Event Query Processor designed specifically to help solve our specific problem discussed previously. After that in chapter 6 we discuss the code of the Engine which we have created in detail with outlining the basic structure and function of the Engine. Following this in Chapter 7 we discuss an instance of a real world problem and its simulated solution on our Engine. And finally we conclude in with some concrete remarks on the evolution of this new field, and a summary of directions of further work that can be done and extended over our Engine.

Chapter 2

Traditional Database and Stream Database

Here in this chapter we will outline some invaluable difference between normal and Stream Databases. Traditional database management systems (DBMSs) expect all data to be managed within some form of persistent data sets. For many recent applications, the concept of a continuous data stream is more appropriate than a data set. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, and others. In all of the applications cited above, it is not feasible to simply load the arriving data into a traditional database management system (DBMS) and operate on it there. Traditional DBMS's are not designed for rapid and continuous loading of individual data items, and they do not directly support the continuous queries that are typical of data stream applications. So stored data set is appropriate when significant portions of the data are queried again and again, and updates are small and/or relatively infrequent. In contrast, a data stream is appropriate when the data is changing constantly (often exclusively through insertions of new elements), and it is either unnecessary or impractical to operate on large portions of the data multiple times. Today's database systems are ill-equipped to perform any kind of special storage management or query processing for data streams, heavily stream-oriented applications tend to use a DBMS largely as an offline storage system, or not at all. Queries over continuous data streams have much in common with queries in a traditional database management system. However, there are two important distinctions peculiar to the data stream model.

The first distinction is between one-time queries and continuous queries . One-time queries (a class that includes traditional DBMS queries) are queries that are evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user. Continuous queries, on the other hand, are evaluated continuously as data streams continue to arrive. The answer to a continuous query is produced over time, always reflecting the stream data seen so far. Continuous query answers may be stored and updated as new data arrives, or they may be produced as data streams themselves.

The second distinction is between predefined queries and ad hoc queries. A predefined query is one that is supplied to the data stream management system before any relevant data has arrived. Predefined queries are generally continuous queries, although scheduled

one-time queries can also be predefined. Ad hoc queries, on the other hand, are issued online after the data streams have already begun. Ad hoc queries can be either one-time queries or continuous queries. Ad hoc queries complicate the design of a data stream management system, both because they are not known in advance for the purposes of query optimization, identification of common sub expressions across queries, etc., and more importantly because the correct answer to an ad hoc query may require referencing data elements that have already arrived on the data streams (and potentially have already been discarded). In the data stream model, some or all of the input data that are to be operated on are not available for random access from disk or memory, but rather arrive as one or more continuous data streams. Data streams differ from the conventional stored relation model in several ways:

- The data elements in the stream arrive online.
- The system has no control over the order in which data elements arrive to be processed, either within a data stream or across data streams.
- Data streams are potentially unbounded in size.
- Once an element from a data stream has been processed it is discarded or archived. It cannot be retrieved easily unless it is explicitly stored in memory, which typically is small relative to the size of the data streams.

Chapter 3

Event Stream Processing and Stream Database

In the Event Stream Processing model, some or all of the input data that are to be operated on are not available for random access from disk or memory, but rather arrive as one or more continuous data streams. Data streams differ from the conventional stored relation model in several ways:

- The data elements in the stream arrive online.
- The system has no control over the order in which data elements arrive to be processed, either within a data stream or across data streams.
- Data streams are potentially unbounded in size.
- Once an element from a data stream has been processed it is discarded or archived. It cannot be retrieved easily unless it is explicitly stored in memory, which typically is small relative to the size of the data streams.

Queries over continuous data streams have much in common with queries in a traditional database management system. However, there are two important distinctions peculiar to the data stream model. The first distinction is between one-time queries and continuous queries [DTO92]. One-time queries (a class that includes traditional DBMS queries) are queries that are evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user. Continuous queries, on the other hand, are evaluated continuously as data streams continue to arrive. Continuous queries are the more interesting class of data stream queries, and it is to them that we will devote most of our attention. The answer to a continuous query is produced over time, always reflecting the stream data seen so far. Continuous query answers may be stored and updated as new data arrives, or they may be produced as data streams themselves. Sometimes one or the other mode is preferred. For example, aggregation queries may involve frequent changes to answer tuples, dictating the stored approach, while join queries are monotonic and may produce rapid, unbounded answers, dictating the stream approach.

The second distinction is between predefined queries and ad hoc queries. A predefined query is one that is supplied to the data stream management system before any relevant

data has arrived. Predefined queries are generally continuous queries, although scheduled one-time queries can also be predefined. Ad hoc queries, on the other hand, are issued online after the data streams have already begun. Ad hoc queries can be either one-time queries or continuous queries. Ad hoc queries complicate the design of a data stream management system, both because they are not known in advance for the purposes of query optimization, identification of common sub expressions across queries, etc., and more importantly because the correct answer to an ad hoc query may require referencing data elements that have already arrived on the data streams (and potentially have already been discarded).

Examples that require a data stream system can be found in many application domains including finance, web applications, security, networking, and sensor monitoring like: web-based financial search engine that evaluates queries over real-time streaming financial data such as stock tickers and news feeds; large web sites monitor web logs (click streams) online to enable applications such as personalization, performance monitoring, and load-balancing (e.g., Yahoo); there are several emerging applications in the area of sensor monitoring [DCZ02] where a large number of sensors are distributed in the physical world and generate streams of data that need to be combined, monitored, and analyzed.A good example of to exploit the Data Stream researchers at Stanford have designed a software called STREAM (STanford stREam datA Manager) [str]. Following is a high-level view of STREAM shown in Figure below.

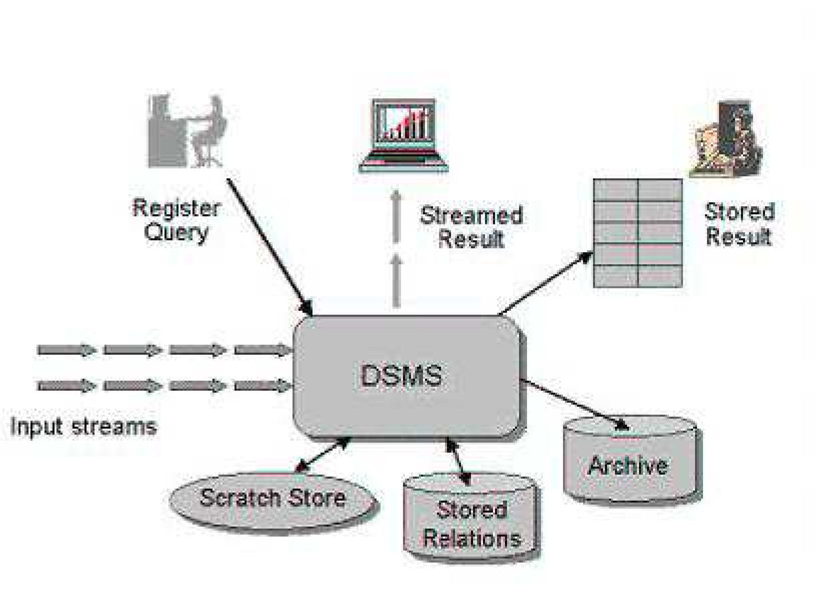

Fig. 3.1 Stream Database structure

In the figure on the left are the incoming Input Streams, which produce data indefinitely and drive query processing. Processing of continuous queries typically requires intermediate state, which is denoted as Scratch Store in the figure. This state could be stored and accessed in memory or on disk. Although it is concerned primarily with the online processing of continuous queries, in many applications stream data may be copied to an Archive, too, for preservation and possible offline processing of expensive analysis or mining queries.

Across the top of the figure the users or applications register Continuous Queries, which remain active in the system until they are explicitly deregistered. Results of continuous queries are generally transmitted as output data streams, but they could also be relational results that are updated over time.

STREAM offers a Web system interface through direct HTTP. To allow interactive use of the system, it has a Web-based GUI as a way to register queries and view results, and we provide an interactive interface for visualizing and modifying system behavior.

3.1 Queries over Data Stream

Query processing in the data stream model of computation comes with its own unique problems. In this section, we will outline what we consider to be the most interesting of these challenges, and describe several alternative approaches available for resolving them. The issues raised in this section will frame a vital part of discussion in the rest of the report.

3.1.1 Unbounded Memory Requirements

Since data streams are potentially unbounded in size, the amount of storage required to compute an exact answer to a data stream query may also grow without bound. While external memory algorithms for handling data sets larger than main memory have been studied, such algorithms are not well suited to data stream applications since they do not support continuous queries and are typically too slow for real-time response. The continuous data stream model is most applicable to problems where timely query responses are important and there are large volumes of data that are being continually produced at a high rate over time. New data is constantly arriving even as the old data is being processed; the amount of computation time per data element must be low, or else the latency of the computation will be too high and the algorithm will not be able to keep pace with the data stream. For this reason, we are interested in algorithms that are able to confine themselves to main memory without accessing disk.

3.1.2 Approximate Query Answering

As described in the previous section, when we are limited to a bounded amount of memory it is not always possible to produce exact answers for data stream queries; however, high-quality approximate answers are often acceptable in lieu of exact answers. Approximation algorithms for problems defined over data streams has been a fruitful research area in the algorithms community in recent years, as discussed in detail in Section 6. This work has led to some general techniques for data reduction and synopsis construction, including: sketches [NAS96], random sampling [SAP00], histograms [IP99] etc. Based on these summarization techniques, we have scoured through some work on approximate query answering. For example, recent work [IP99] develops histogram-based techniques to provide approximate answers for correlated aggregate queries over data streams, and present a general approach for building small space summaries over data streams to provide approximate answers for many classes of aggregate queries. However, research problems abound in the area of approximate query answering, with or without streams. Even the basic notion of

approximations remains to be investigated in detail for queries involving more than simple aggregation. In the next two subsections, we will go through several approaches proposed by peers in this research area to approximation, some of which are peculiar to the data stream model of computation.

3.1.3 Sliding Windows

One technique for producing an approximate answer to a data stream query is to evaluate the query not over the entire past history of the data streams, but rather only over sliding windows of recent data from the streams. For example, only data from the last week could be considered in producing query answers, with data older than one week being discarded. Imposing sliding windows on data streams is a natural method for approximation that has several attractive properties. It is well-defined and easily understood: the semantics of the approximation are clear, so that users of the system can be confident that they understand what is given up in producing the approximate answer. It is deterministic, so there is no danger that unfortunate random choices will produce a bad approximation. Most importantly, it emphasizes recent data, which in the majority of real-world applications is more important and relevant than old data: if one is trying in real-time to make sense of network traffic patterns, or phone call or transaction records, or scientific sensor data, then in general insights based on the recent past will be more informative and useful than insights based on stale data. In fact, for many such applications, sliding windows can be thought of not as an approximation technique reluctantly imposed due to the infeasibility of computing over all historical data, but rather as part of the desired query semantics explicitly expressed as part of the user's query. For example queries which just require the recent data arrived don't specifically deal with older data (one example is calculating the maximum sale per minute registered so far in the stock market, this only needs the previous data.)

There are a variety of research issues in the use of sliding windows over data streams. To begin with, as we will discuss later, there is the fundamental issue of how one defines timestamps over the streams to facilitate the use of windows. Extending SQL or relational algebra to incorporate explicit window specifications is nontrivial. The implementation of sliding window queries and their impact on query optimization is a largely untouched area. In the case where the sliding window is large enough so that the entire contents of the window cannot be buffered in memory, there are also theoretical challenges in designing algorithms that can give approximate answers using only the available memory. Some recent results in this vein can be found in [**?**].

While existing work on sequence and temporal databases has addressed many of the issues involved in time-sensitive queries (a class that includes sliding window queries) in a relational database context, differences in the data stream computation model pose new challenges. Research in temporal databases is concerned primarily with maintaining a full history of each data value over time, while in a data stream system we are concerned primarily with processing new data elements on-the-fly. Sequence databases attempt to produce query plans that allow for stream access, meaning that a single scan of the input data is sufficient to evaluate the plan and the amount of memory required for plan evaluation is a constant, independent of the data. This model assumes that the database system

has control over which sequence to process tuples from next, e.g., when merging multiple sequences, which we cannot assume in a data stream system.

3.1.4 Batch Processing, Sampling, and Synopses

Another class of techniques for producing approximate answers is to give up on processing every data element as it arrives, resorting to some sort of sampling or batch processing technique to speed up query execution. We describe a general framework for these techniques. Suppose that a data stream query is answered using a data structure that can be maintained incrementally. The most general description of such a data structure is that it supports two operations, update(tuple) and computeAnswer(). The update operation is invoked to update the data structure as each new data element arrives, and the computeAnswer method produces new or updated results to the query. When processing continuous queries, the best scenario is that both operations are fast relative to the arrival rate of elements in the data streams. In this case, no special techniques are necessary to keep up with the data stream and produce timely answers: as each data element arrives, it is used to update the data structure, and then new results are computed from the data structure, all in less than the average inter-arrival time of the data elements. If one or both of the data structure operations are slow, however, then producing an exact answer that is continually up to date is not possible. One has to consider the two possible bottlenecks and approaches for dealing with them which are Batch Processing and Sampling.

3.1.5 Blocking Operators

A blocking query operator is a query operator that is unable to produce the first tuple of its output until it has seen its entire input. Sorting is an example of a blocking operator, as are aggregation operators such as SUM, COUNT, MIN, MAX, and AVG. If one thinks about evaluating continuous stream queries using a traditional tree of query operators, where data streams enter at the leaves and final query answers are produced at the root, then the incorporation of blocking operators into the query tree poses problems. Since continuous data streams may be infinite, a blocking operator that has a data stream as one of its inputs will never see its entire input, and therefore it will never be able to produce any output. Clearly, blocking operators are not very suitable to the data stream computation model, but aggregate queries are extremely common, and sorted data is easier to work with and can often be processed more efficiently than unsorted data. Doing away with blocking operators altogether would be problematic, but dealing with them effectively is one of the more challenging aspects of data stream computation. Blocking operators that are the root of a tree of query operators are more tractable than blocking operators that are interior nodes in the tree, producing intermediate results that are fed to other operators for further processing (for example, the "sort" phase of a sort-merge join, or an aggregate used in a subquery). When we have a blocking aggregation operator at the root of a query tree, if the operator produces a single value or a small number of values, then updates to the answer can be streamed out as they are produced. When the answer is larger, however, such as when the query answer is a relation that is to be produced in sorted order, it is more practical to maintain a data structure with the up-to-date answer, since continually

retransmitting the entire answer would be cumbersome. Neither of these two approaches works well for blocking operators that produce intermediate results, however. The central problem is that the results produced by blocking operators may continue to change over time until all the data has been seen, so operators that are consuming those results cannot make reliable decisions based on the results at an intermediate stage of query execution.

3.1.6 Queries Referencing Past Data

In the data stream model of computation, once a data element has been streamed by, it cannot be revisited. This limitation means that ad hoc queries that are issued after some data has already been discarded may be impossible to answer accurately. One simple solution to this problem is to stipulate that ad hoc queries are only allowed to reference future data: they are evaluated as though the data streams began at the point when the query was issued and any past stream elements are ignored (for the purposes of that query). While this solution may not appear very satisfying, it may turn out to be perfectly acceptable for many applications.

A more ambitious approach to handling ad hoc queries that reference past data is to maintain summaries of data streams (in the form of general-purpose synopses or aggregates) that can be used to give approximate answers to future ad hoc queries. Taking this approach requires making a decision in advance about the best way to use memory resources to give good approximate answers to a broad range of possible future queries. The problem is similar in some ways to problems in physical database design such as selection of indexes and materialized views [CN97] However, there is an important difference: in a traditional database system, when an index or view is lacking, it is possible to go to the underlying relation, albeit at an increased cost. In the data stream model of computation, if the appropriate summary structure is not present, then no further recourse is available.

3.2 Query Language for a DSMS

Any general-purpose data management system must have a flexible and intuitive method by which the users of the system can express their queries. In the STREAM project, they have chosen to use a modified version of SQL as the query interface to the system (although they are also providing a means to submit query plans directly). SQL is a well-known language with a large user population. It is also a declarative language that gives the system flexibility in selecting the optimal evaluation procedure to produce the desired answer. Other methods for receiving queries from users are possible; for example, the Aurora system described in [DCZ02] uses a graphical "boxes and arrows" interface for specifying data flow through the system. This interface is intuitive and gives the user more control over the exact series of steps by which the query answer is obtained than is provided by a declarative query language.

The main modification that they have made to standard SQL, in addition to allowing the FROM clause to refer to streams as well as relations, is to extend the expressiveness of the query language for sliding windows. It is possible to formulate sliding window queries in SQL by referring to timestamps explicitly, but it is often quite awkward. SQL-99 introduces

analytical functions that partially address the shortcomings of SQL for expressing sliding window queries by allowing the specification of moving averages and other aggregation operations over sliding windows. However, the SQL-99 syntax is not sufficiently expressive for data stream queries since it cannot be applied to non-aggregation operations such as joins.

The notion of sliding windows requires at least an ordering on data stream elements. In many cases, the arrival order of the elements suffices as an "implicit timestamp" attached to each data element; however, sometimes it is preferable to use "explicit timestamps" provided as part of the data stream. Formally we say (following [DCZ02]) that a data stream : consists of a set of (tuple, timestamp) pairs. The timestamp attribute could be a traditional timestamp or it could : be a sequence number - all that is required is that it comes from a totally ordered domain with a distance metric. The ordering induced by the timestamps is used when selecting the data elements making up a sliding window.

They extend SQL by allowing an optional window specification to be provided, enclosed in brackets, after a stream (or subquery producing a stream) that is supplied in a query's FROM clause. A window specification consists of:

1. An optional partitioning clause, which partitions the data into several groups and maintains a separate window for each group,
2. A window size, either in "physical" units (i.e., the number of data elements in the window) or in "Logical" units (i.e., the range of time covered by a window, such as 30 days), and
3. An optional filtering predicate.

3.3 Timestamps in Streams

In it sliding windows are defined with respect to a timestamp or sequence number attribute representing a tuple's arrival time. This approach is unambiguous for tuples that come from a single stream, but it is less clear what is meant when attempting to apply sliding windows to composite tuples that are derived from tuples from multiple underlying streams (e.g., windows on the output of a join operator). What should the timestamp of a tuple in the join result be when the timestamps of the tuples that were joined to form the result tuple are different? Timestamp issues also arise when a set of distributed streams make up a single logical stream, as in the web monitoring application described in previous sections, or in truly distributed streams such as sensor networks when comparing timestamps across stream elements may be relevant.

In the previous section we talked about implicit timestamps, in which the system adds a special field to each incoming tuple, and explicit timestamps, in which a data attribute is designated as the timestamp. Explicit timestamps are used when each tuple corresponds to a real-world event at a particular time that is of importance to the meaning of the tuple. Implicit timestamps are used when the data source does not already include timestamp information, or when the exact moment in time associated with a tuple is not important, but general considerations such as "recent" or "old" may be important.

Chapter 4

Our Aim, the System and the Problem

Here we will discuss about what we are trying to achieve i.e. our aim, then we will discuss about the system or the work environments with which we have to work and on which we have to judge the correctness of our Engine.

4.1 Our Aim

Initially we wanted to solve the Linear Road Problem which is a common Bench Mark for any stream or event driven database processor. In simple terms Linear Road Problem is described as follows. Variable tolling refers to charging a vehicle different toll rates based on the time of day or level of congestion of a roadway. Linear Road [?] is inspired by the increasing prevalence of variable tolling on highway systems in cities throughout the world. Linear Road specifies a variable tolling system for a fictional urban expressway system where tolls are determined based on changing factors such as congestion and accident proximity. Each car on the expressway is equipped with a transponder or sensor that emits a position report that identifies the vehicle's exact location (coordinates) every 30 seconds. These position reports are used to generate statistics about traffic conditions on every segment of every expressway for every minute. These statistics, including average vehicle speed, number of vehicles and existence of accidents, are used to determine toll charges for the given segment of road that the vehicle is in. This tolling is designed to control the traffic flow roadways by discouraging drivers from using already congested roads through increased tolls. Alternatively, this may encourage the use of less congested roads through decreased tolls.

The purpose of the benchmark is to determine the performance metric of a stream processing system: the maximum scale at which the system can respond to the specified set of continuous and historical queries while meeting their response time and accuracy requirements. It is assumed that the benchmark will run with increasingly larger scale factors until one is found for which the requirements cannot be met. Linear Road tests Stream Data Management Systems (SDMS) by measuring how many expressways a SDMS can support by giving accurate and timely results to four types of queries that fit two

Fig. 4.1 Waiting in line for toll booths is starting to become a thing of the past as variable tolling begins to emerge and better technology is developed.

categories: continuous and historical. To run the benchmark, available in the download area, generates data for a selected number of expressways. This data is for a 2 hour span of vehicles traveling on an expressway(s). The stream data generated by the simulator consists of four types of tuples (queries): Position Reports and historical query requests for Account Balances, Daily Expenditures and Travel Time Estimation. Toll Notification, triggered by a position report, tells the vehicle of the toll it will be accessed for entering the next segment of the expressway it is traveling on. Accident Notification, an accident occurs when two vehicles are stopped at the same position at the same time. A vehicle is stopped when it reports the same position in 4 consecutive position reports. Once an accident occurs in a given segment, traffic proceeds in that segment at a reduced speed determined by the traffic spacing model. The accident takes anywhere from 10-20 minutes to be cleared once it is detected. Accidents effect the amount tolled to a car. Account Balance Query is an account request for a vehicle. It returns the total amount in an account for any driver that requests it. A Daily Expenditures Query is a request for a vehicle's total tolls on a specific expressway, on a specified day in the previous ten weeks. Travel Time Estimation is a request for an estimated toll and travel time for a journey on a given expressway, day of the week and time of day.

So for that we had to design an engine for collecting and evaluating stream events. As a result "Disang" our event processing Engine was created. This is a basic event processing engine made with the help of Eclipse Platform and written in java language.

4.2 The System

We are basically working with a dynamic system which consists of following three main components, A main central control system (call it engine) which processes all the incoming events, a set of nodes or senders which send events to the central unit or the engine and a vital third component is the communication channel between these two units. There are several real life examples of such systems like Sensor nodes and the central node or the base station of the network and the communication network between them. Or an airline might process event feeds of flight positions and weather, monitoring, constantly analyzing and

looking for conditions that provoke action, such as to propose a new flight route or rebook a passenger. As in figure

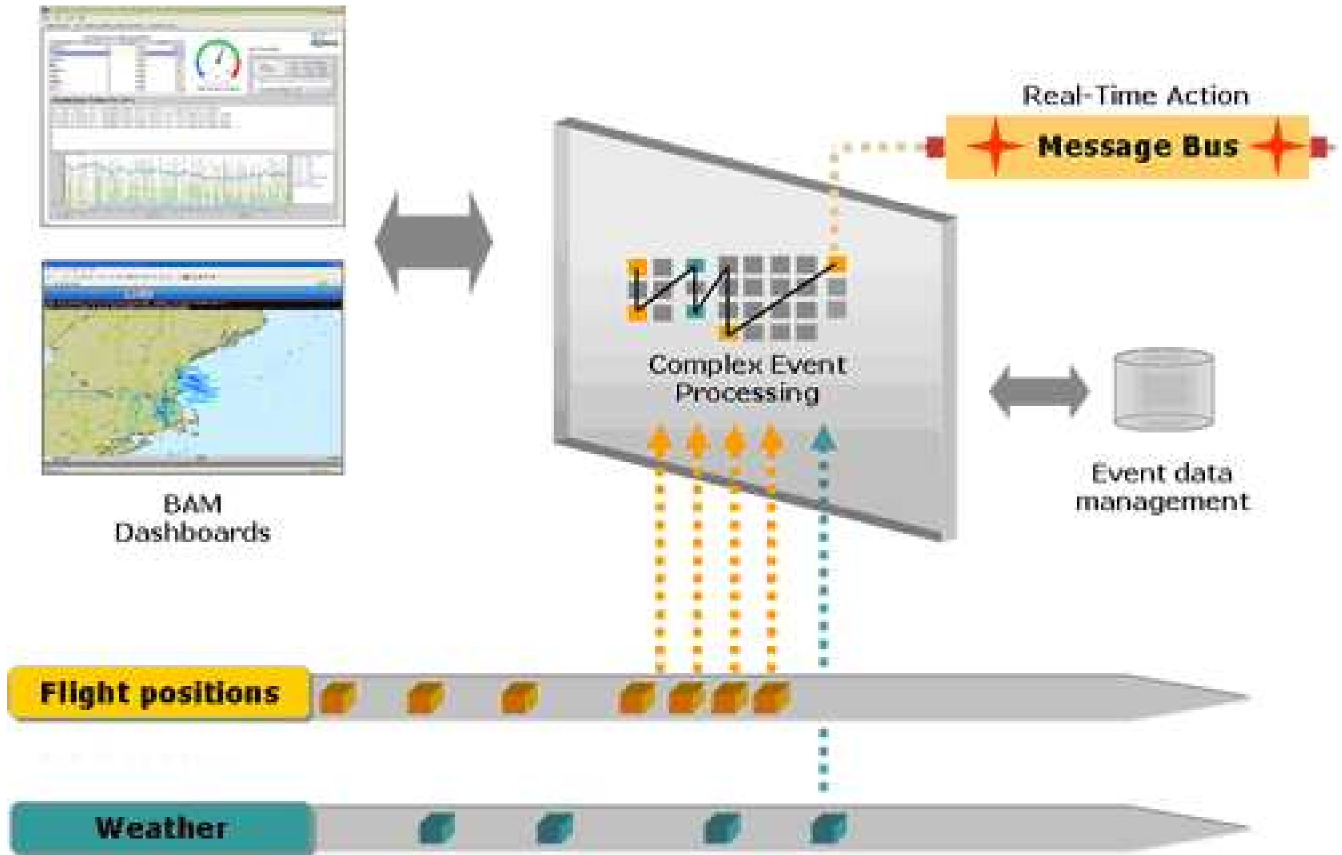

Fig. 4.2 Flight Management System

One major instance of such system and which we are going to discuss in detail in later chapters and to which we will put our engine to use is A Self-Service Terminal Managing System. A self-service terminal system as it exists in airports is to allow customers to proceed to self-check in and print boarding passes. The self-service terminal managing system gets a lot of events from all the connected terminals. The event rate is around 500 events per second. Some events indicate abnormal situations such as "paper low" or "terminal out of order." Other events observe activity as a customer uses a terminal to check in and print her boarding pass. A self-service terminal system as it exists in airports to allow customers to proceed to self-check in and print boarding passes. The self-service terminal managing system gets a lot of events from all the connected terminals. The event rate is around 500 events per second. Some events indicate abnormal situations such as "paper low" or "terminal out of order." Other events observe activity as a customer uses a terminal to check in and print her boarding pass.

Our primary goal is to resolve self-service terminal or network problems before our customers report them by looking for help, which means higher overall availability and greater customer satisfaction. To accomplish this, we would like to get alerted when certain conditions occur that warrant human intervention: for example, a customer may be in the

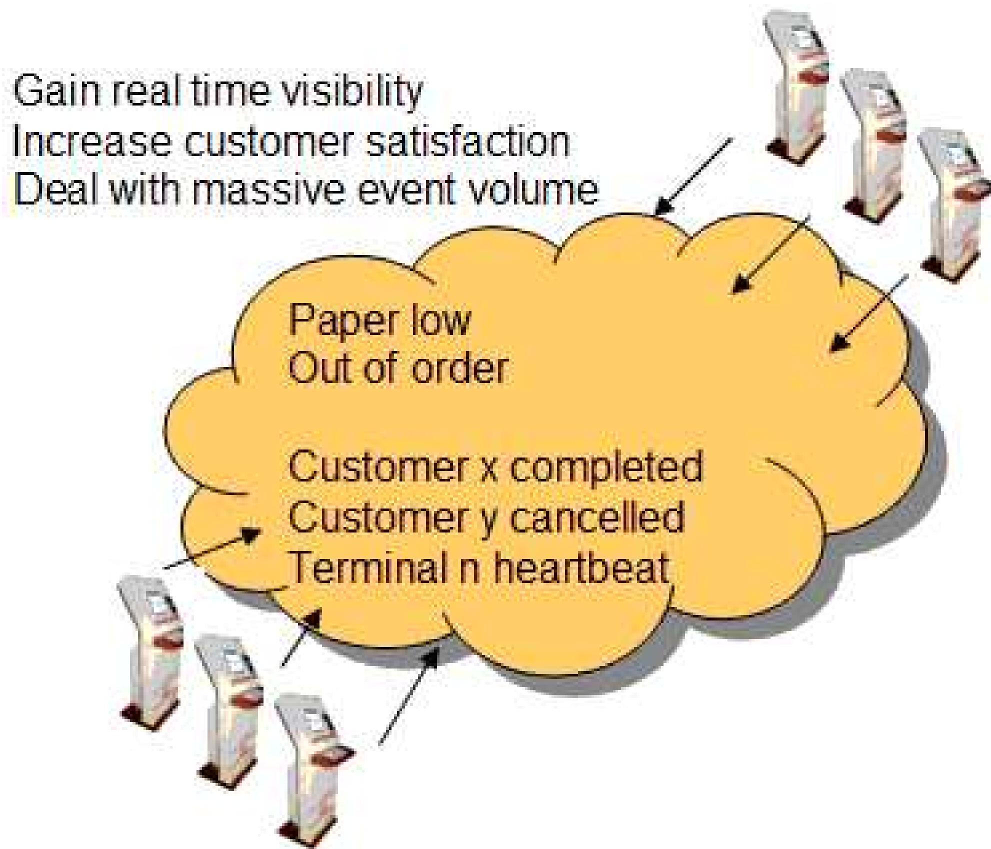

Fig. 4.3 Event cloud in a terminal managing system

middle of a check-in process when the terminal detects a hardware problem or when the network goes down. Under these conditions we would like to dispatch a staff member to help that customer, and another staff member to diagnose the hardware or network problem. We also want to provide a dashboard and summarize activity on an ongoing basis and feed this to a real-time interface. This enables a manager to watch the system in action and spot abnormalities. The system can further compare the summarized activity to stored normal usage patterns.

4.3 The Problem

Our problem is to solve a real life stream database scenario or using our Engine to process an Event stream. As we have described above about Self-Service Terminal Managing System. We will solve this Self-Service Terminal Managing System problem using our Engine. So for solving this problem we have to design an engine with following properties, First of all it must have a logging mechanism so that the code can be debugged easily if some problem arises it must take events as input, it must also register queries which are a kind of flags which will differentiate between various incoming events. So for registering queries it must also have a parser so that our queries have a format and can be easily interpreted by the system. We need listeners as well for a query so that if a desired event occurs then it is reported to the listener. So we need a parser as well. It must also have a filter which will filter out the desired events from the irrelevant ones. Also we must have a scheduler which will keep track of any queues and how they are operated in between themselves. We must also have an internal daemon Timer for the engine which will keep track of the timing of the incoming events. Apart from that we must have scheduling buckets to each Query. We must also have views representation scheme within a query. Finally we must also have Dispatch services to inform the listeners to a Query statement about the occurrence of a desired event. So these all things are being provided in our Engine in order to cope with the demands of the problem.

Chapter 5

Previous Works

Its a relatively new and burgeoning area of research and thus there has been very few literature in this are. But there are few projects which are worth mentioning as these provide the backbone of our own Stream Engine. We will outline the basic structure of few of these projects.

5.1 STREAM (Stanford Stream Database Manager)

The STanford stREam datA Manager (STREAM) project at Stanford is a general-purpose Data Stream Management System (DSMS) for processing continuous queries over multiple continuous data streams and stored relations. There are two following fundamental differences between a DSMS and a traditional DBMS:

1. A DSMS must handle multiple continuous, high volume, and possibly time-varying data streams in additional to managing traditional stored relations.

2. Due to the continuous nature of data streams, a DSMS needs to support long-running continuous queries, producing answers in a continuous and timely fashion.

STREAM supports declarative continuous queries over two types of inputs: streams and relations. A continuous query is simply a long-running query, which produces output in a continuous fashion as the input arrives. The queries are expressed in a language called CQL. The input types-streams and relations are defined using some ordered time domain, which may or may not be related to wall-clock time.

A high-level view of STREAM is shown in Figure above. On the left are the incoming Input Streams, which produce data indefinitely and drive query processing. Processing of continuous queries typically requires intermediate state, which we denote as Scratch Store in the figure. This state could be stored and accessed in memory or on disk. Although we are concerned primarily with the online processing of continuous queries, in many applications stream data also may be copied to an Archive, for preservation and possible offline processing of expensive analysis or mining queries. Across the top of the figure we see that users or applications register Continuous Queries, which remain active in the system until they are explicitly deregistered. Results of continuous queries are generally transmitted as output data streams, but they could also be relational results that are updated over

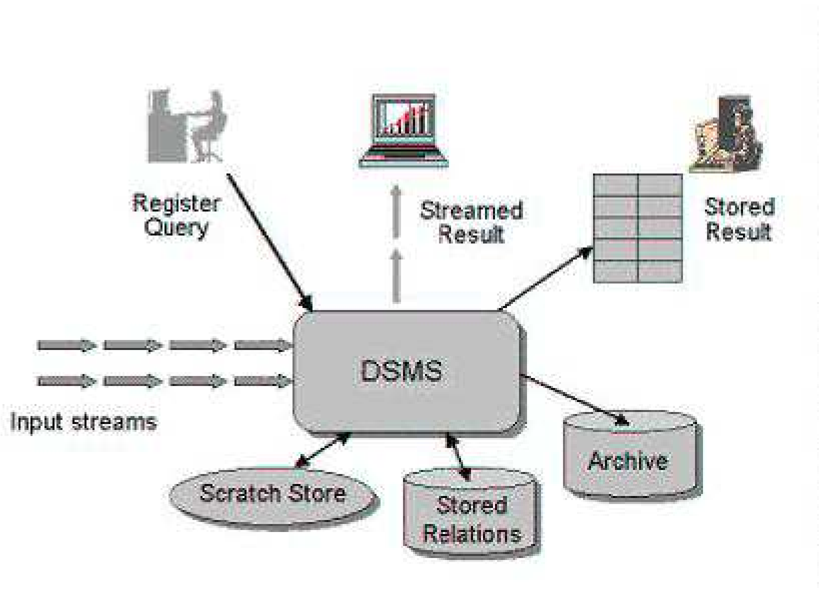

Fig. 5.1 Stream Database structure

time.STREAM offers a Web system interface through direct HTTP. To allow interactive use of the system, it has a Web-based GUI as a way to register queries and view results, and we provide an interactive interface for visualizing and modifying system behavior.

The CQL Continuous Query Language: This Query language was developed by STREAM people as a use in their own Event processing engine. For simple continuous queries over streams, it can be sufficient to use a relational query language such as SQL, replacing references to relations with references to streams, and streaming new tuples in the result. However, as continuous queries grow more complex, e.g., with the addition of aggregation, sub queries, windowing constructs, and joins of streams and relations, the semantics of a conventional relational language applied to these queries quickly becomes unclear. To address this problem, we have defined a formal abstract semantics for continuous queries, and we have designed CQL, a concrete declarative query language that implements the abstract semantics.

5.1.1 Abstract Semantics

The abstract semantics is based on two data types, streams and relations, which are defined using a discrete, ordered time domain τ

A stream S is an unbounded bag (multiset) of pairs <s, z >, where s is a tuple and $z \in R$ is the timestamp that denotes the logical arrival time of tuple s on stream S.

A relation R is a time - varying bag of tuples. The bag of tuples at time $z \in R$ is denoted R(z), and we call R(z) an instantaneous relation. Note that our definition of a relation differs from the traditional one which has no built-in notion of time.

The abstract semantics uses three classes of operators over streams and relations:

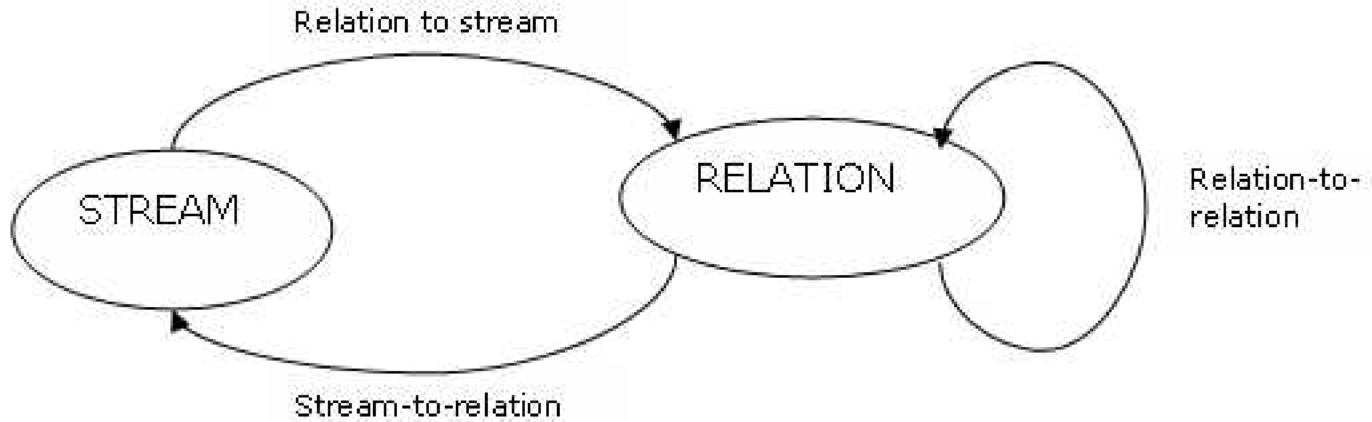

Fig. 5.2 Data types and operator classes in abstract semantics.

- A relation-to-relation operator takes one or more relations as input and produces a relation as output.
- A stream-to-relation operator takes a stream as input and produces a relation as output.
- A relation-to-stream operator takes a relation as input and produces a stream as output.

Stream-to-stream operators are absent. They are composed from operators of the above three classes. These three classes are "black box" components of our abstract semantics: the semantics does not depend on the exact operators in these classes, but only on generic properties of each class. Figure 2 summarizes our data types and operator classes.Our concrete declarative query language, CQL (for Continuous Query Language), is defined by instantiating the operators of our abstract semantics. Syntactically, CQL is a relatively minor extension to SQL.

5.1.2 Relation-to-Relation Operators in CQL

CQL uses SQL constructs to express its relation-to-relation operators, and much of the data manipulation in a typical CQL query is performed using these constructs, exploiting the rich expressive power of SQL.

5.1.3 Relation-to-Stream Operators in CQL

CQL has three relation-to-stream operators: Istream, Dstream, and Rstream.

Istream (for "insert stream") applied to a relation R contains <s, r >whenever tuple s is in R(r) - R(r-1), i.e., whenever s is inserted into R at time r.

Dstream (for "delete stream") applied to a relation R contains <s, r >whenever tuple s is in R(r-1) - R(r), i.e., whenever s is deleted from R at time r.

Rstream (for "relation stream") applied to a relation R contains the <s, r >whenever tuple s is in R(r), i.e., every current tuple in R is streamed at every time instant.

5.1.4 Query Plans and Execution

When a continuous query specified in CQL is registered with the STREAM system, a query plan is compiled from it. Query plans are composed of operators, which perform the actual processing, queues, which buffer tuples (or references to tuples) as they move between operators, and synopses, which store operator state.

5.1.5 Operator

Query plan operator reads from one or more input queues, processes the input based on its semantics, and writes any output to an output queue. Individual operators may materialize their relational inputs in synopses if such state is useful. During execution, operators are scheduled individually, allowing for fine grained control over queue sizes and query latencies.

5.1.6 Queue

A queue in a query plan connects its producing plan operator Op to its consuming operator Oc. At any time a queue contains a (possibly empty) collection of elements representing a portion of a stream or relation. The elements that Op produces are inserted into the queue and buffered there until they are processed by Oc. Operators in our system requires elements on their input queues be read in non decreasing timestamp order.

5.1.7 Synopses

Synopsis belongs to a specific plan operator, storing state that may be required for future evaluation of that operator. Synopses are shared among operators to optimize the continuous query. The most common use of a synopsis in this system is to materialize the current state of a (derived) relation, such as the contents of a sliding window or the relation produced by a sub query. Synopses are also used to store a summary of the tuples in a stream or relation for approximate query. Synopses (and queues) are kept in memory.

5.1.8 Query Plan Execution

When a query plan is executed, a scheduler selects operators in the plan to execute in turn. The semantics of each operator depends only on the timestamps of the elements

it processes, not on system or "wall-clock" time. Thus, the order of execution has no effect on the data in the query result, although it can affect other properties such as latency and resource utilization.

5.2 Esper

Esper is a 100% Java component for CEP and ESP applications [esp]. Esper enables rapid development of applications that process large volumes of incoming messages or events. Esper filters and analyzes events in various ways, and responds to conditions of interest in real-time. Using Java in Esper has been a great advantage of this project and thus it has scored miles over other Event Stream Processing Engines like STREAM and few others. This Project uses a query language called EQL (event query language) which is more or less similar to the CQL used by the STREAM people of Stanford. EQL is very similar to SQL in it's syntax and offers additional capabilities for event stream processing. As part of EQL, Esper also offers a pattern language that provides for stateful (state-machine) event pattern matching. But the only problem with this research is that it is not fully open sourced. So their source code and structure of their Engine is not fully available for research work.

5.3 A few other projects

A few other projects are The Aura Project developed jointly by MIT, Brown University and Brandeis University [aur]. The primary goal of the Aurora project is to build a single infrastructure that can efficiently and seamlessly meet the requirements of such demanding applications. To this end, we are currently critically rethinking many existing data management and processing issues, as well as developing new proactive data processing concepts and techniques. Aurora addresses three broad application types in a single, unique framework:

1. Real-time monitoring applications continuously monitor the present state of the world and are, thus, interested in the most current data as it arrives from the environment. In these applications, there is little or no need (or time) to store such data.

2. Archival applications are typically interested in the past. They are primarily concerned with processing large amounts of finite data stored in a time-series repository.

3. Spanning applications involve both the present and past states of the world, requiring combining and comparing incoming live data and stored historical data. These applications are the most demanding as there is a need to balance real-time requirements with efficient processing of large amounts of disk-resident data.

Chapter 6

The Disang Engine

Here we will discuss a sort of skeletal structure of our Disang Event stream processor. Our Engine has many components. We will illustrate all of them one by one. For each component we will first explain their use then will discuss the classes used for these components and if necessary will put some snippets of the code as well. Finally we will discuss how each of these components, their classes and interfaces interact together and finally how is the engine as a whole runs.

6.1 Configuration Component

This is our first component in the engine. It defines the configuration of the events which are to be sent to the engine. Configuration means the parameter of the events. More importantly configuring the events at analyzing a common property reduces the time for debugging. For this we have used the Apache Commons logging parameters and thus I have to use the Log.jar library. Inserting log statements into your code is a low-tech method for debugging it. It may also be the only way because debuggers are not always available or applicable. This is often the case for distributed applications. On the other hand, some people argue that log statements pollute source code and decrease legibility. (We believe that the contrary is true). In the Java language where a preprocessor is not available, log statements increase the size of the code and reduce its speed, even when logging is turned off. Given that a reasonably sized application may contain thousands of log statements, speed is of particular importance. With log4j it is possible to enable logging at runtime without modifying the application binary. The log4j package is designed so that these statements can remain in shipped code without incurring a heavy performance cost. Logging behavior can be controlled by editing a configuration file, without touching the application binary. Logging equips the developer with detailed context for application failures. On the other hand, testing provides quality assurance and confidence in the application. Logging and testing should not be confused. They are complementary. When logging is wisely used, it can prove to be an essential tool. One of the distinctive features of log4j is the notion of inheritance in loggers. Using a logger hierarchy it is possible to control

which log statements are output at arbitrarily fine granularity but also great ease. This helps reduce the volume of logged output and minimize the cost of logging. The target of the log output can be a file, an "OutputStream", a "java.io.Writer", a remote log4j server, a remote Unix "Syslog" daemon, or many other output targets. On an AMD "Duron" clocked at 800Mhz running JDK 1.3.1, it costs about 5 nanoseconds to determine if a logging statement should be logged or not. Actual logging is also quite fast, ranging from 21 microseconds using the "SimpleLayout", 37 microseconds using the "TTCCLayout". The performance of the "PatternLayout" is almost as good as the dedicated layouts, except that it is much more flexible. The Configuration parameters in the Disang engine are inserted using the "ConfigurationSnapShot" Class.

6.2 Scheduling Component

This is used in Scheduling of various view windows. Scheduler is quite important from the point of importance of data streams. It is a sort of sliding window which joins/groups/augments (depending upon the query requirement) different buckets or queues and synchronizes the whole thing. The following figure elaborates the point

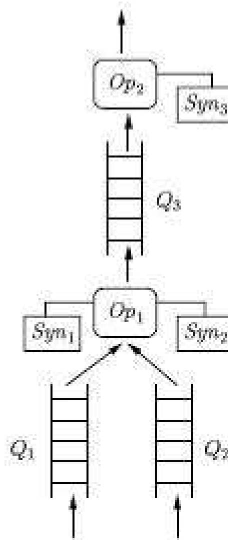

Fig. 6.1 scheduling of different Views

Scheduling services are added to the Engine using "SchedulingServiceProvider" class. It in turn calls "Schedulerserviceimpl" class to initiate these services. It also provides a scheduling bucket per EQL statement (query to be registered) "Schedulebucket" class allocates one schedule bucket for each EQL statement.

6.3 Event Component

Events in Disang are represented via regular Java classes that expose JavaBean-style "getter" methods for access to event properties. Event services are added to the Engine using EventAdapterServices class. Classes like BeanEventAdapter Bean-Eventtype PropertyListBuilder etc further subclassify each event so that looking for a particular pattern becomes easier. Disang is able to handle events as JavaBeans as of now but in future can be extended to arbitrary java classes, java.util.Map, or XML documents. Now for instance pick the self-service terminal system for case study. In this case study we assume we decided to use the JavaBeans representation for simplicity. Each self-service terminal publishes any of the six events kind below.

Checking	A customer started a check-in dialog
Cancelled	A customer cancelled a check-in dialog
Completed	A customer completed a check-in dialog
OutOfOrder	A terminal detected a hardware problem
LowPaper	A terminal is low on paper
Status	Terminal status, published every 1 minute regardless of activity

Table 6.1 Some feasible Events

All events provide information about the terminal that published the event. Since all events carry similar information, we model each event as a subtype to a base class "BaseTerminalEvent", which will provide the terminal information that all events issued by a terminal share. A real-world model would of course be more complex–possibly using XML instead.

```
public abstract class BaseTerminalEvent {
    private final Terminal terminal;

    public BaseTerminalEvent(Terminal terminal) {
        this.terminal = terminal;
    }

    public String getType() {
        return this.getClass().getSimpleName();
    }

    public Terminal getTerminal() {
        return terminal;
    }
}
```

For the terminal information we use a simple class to hold the terminal ID:

```
public class Terminal {
    private String id;

    public Terminal(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }
}
```

The Status event class is thus a subclass of "BaseTerminalEvent":

```
public class Status extends BaseTerminalEvent {
    public Status(Terminal terminal) {
        super(terminal);
    }
}
```

6.4 EQL (Event Query Language) Component

Disang EQL is an object-oriented event stream query language very similar to SQL in its syntax but that significantly differs to be able to deal with sliding window of streams of data. Disang also includes a pattern language that provides for stateful (state-machine) event pattern matching. EQL and patterns can be used alone or can also be combined to express complex temporal logic. Considering the case study of the self service terminal just discussed in the previous section let's assume we want to dispatch staff to restock paper supply when a terminal publishes a LowPaper event. The simple EQL statement below detects such events:

```
select * from LowPaper
```

Besides looking for LowPaper events, we would also like to be notified when OutOfOrder events arrive. We could thus use two separate statements that each filter for only one type of event:

```
select * from LowPaper
select * from OutOfOrder
```

Another solution would be to use a single statement and include an event pattern combined with an or condition:

```
select a,b from pattern [ every a=LowPaper or every b=OutOfOrder]
```

We could also implement this with the help of event polymorphism and the "BaseTerminalEvent". As all events are subclasses of "BaseTerminalEvent", we can use a where clause to filter out the events we are interested in and use the type property (that is the actual class name without package information–see "BaseTerminalEvent"):

```
select * from BaseTerminalEvent
where type = 'LowPaper' or type = 'OutOfOrder'
```

Deciding on a statement merely depends on design choices in this simple case.

6.5 Parser Component

Since we are using our own query semantics thus we require a parser for this purpose. For registering the query the query is first written as a string and is passed to the engine with the help of EPAdmin Class which first passes it to a parser. For parsing purpose a builtin ANTLR Tree Parser is used. For that we have used a standard ANTLR library. Parsing is the application of grammatical structure to a stream of input symbols. ANTLR takes this further than most tools and considers a tree to be a stream of nodes, albeit in two dimensions. In fact, the only real difference in ANTLR's code generation for token stream parsing versus tree parsing lies in the testing of lookahead, rule-method definition headers, and the introduction of a two-dimensional tree structure code-generation template. ANTLR tree parsers can walk any tree that implements the AST interface, which imposes a child-sibling like structure to whatever tree data-structure you may have. The important navigation methods are:

- `getFirstChild`:

  Return a reference to the first child of the sibling list.

- `getNextSibling`:

  Return a reference to the next child in the list of siblings.

Each AST node is considered to have a list of children, some text, and a "token type". Trees are self-similar in that a tree node is also a tree. An AST is defined completely as:

```java
/** Minimal AST node interface used by ANTLR AST generation
 * and tree-walker.
 */
public interface AST {
    /** Add a (rightmost) child to this node */
    public void addChild(AST c);
    public boolean equals(AST t);
    public boolean equalsList(AST t);
    public boolean equalsListPartial(AST t);
    public boolean equalsTree(AST t);
    public boolean equalsTreePartial(AST t);
    public ASTEnumeration findAll(AST tree);
    public ASTEnumeration findAllPartial(AST subtree);
    /** Get the first child of this node; null if no children */
    public AST getFirstChild();
    /** Getthe next sibling in line after this one */
    public AST getNextSibling();
    /** Get the token text for this node */
    public String getText();
    /** Get the token type for this node */
    public int getType();
    /** Get number of children of this node; if leaf, returns 0 */
    public int getNumberOfChildren();
    public void initialize(int t, String txt);
    public void initialize(AST t);
    public void initialize(Token t);
    /** Set the first child of a node. */
    public void setFirstChild(AST c);
    /** Set the next sibling after this one. */
    public void setNextSibling(AST n);
    /** Set the token text for this node */
    public void setText(String text);
    /** Set the token type for this node */
    public void setType(int ttype);
    public String toString();
    public String toStringList();
    public String toStringTree();
}
```

6.6 Miscellaneous other Components

There are few other critical service components which are important for the well processing of the incoming Event stream by the engine. We will mention those in this section.

6.6.1 AutoImport Services

In order for user to import some java packages we have to create "AutoImportServiceImpl" class. This class can enable user to import java services to user for the purpose of interaction with the engine.

6.6.2 Database Services

Then we had to add some usual database services in the engine and for this purpose we created "DatabaseConfigServiceImpl" class.

6.6.3 Timer Services

This service is very crucial as it creates a sort of timing mechanism for the Engine, using which the engine keeps track of all the incoming events and related timestamps. Also when the timer service is added a deamon (background thread) timer is started which notifies Engine of critical events and also used in Callback when any matched pattern is reported by the engine.

6.6.4 Emit Services

This service is for attaching listeners(those to be notified in case of pattern matching) to the EQL statements.

6.6.5 Dispatch Services

This service is for informing the listeners in case of any match of pattern with in the incoming events.

6.6.6 View Services

This service is for simulating Views just like in case of normal databases.

6.6.7 Stream Services

This service is for controlling stream of data like time windows, queues etc

6.6.8 Read-Write Lock Services

This service is for modifying views and windows etc at runtime in multithreaded environment. Since this needs synchronization, that is why we need a locking mechanism.

6.6.9 Runtime coordination Services

This service is for runtime coordination between various services

6.6.10 Administrative Services

This service is for the administrative control of the engine.

There are plenty of more classes and services whose function is a hybrid of all these above services and listing those above will unnecessarily make the picture ambiguous. We will discuss them in the next section when we will see how the engine works and gets initialized.

6.7 Initialization and the Working of the Engine

First in the simulation program (one which generates the batch of events or so to say the data stream, and external agent can initialize its own Disang Engine as well) we initialize the engine: It is done as follows:

First the Configuration of the events which are to be sent to the engine are defined. Configuration means the parameter of the events. For Defining the configuration I have used the Apache Commons logging parameters and thus I have to use the Log.jar library for running our code. After adding the entire configuration I call the class "EPServiceProviderManager" which in turn calls "EPServiceProviderImpl" and passes the Configuration parameter to it. The EPServiceProviderImpl" class first puts the Configuration parameter in the Disang engine using the "ConfigurationSnapShot" interface. Then it goes on to add required services to the engine.

First it adds scheduling services using "SchedulingServiceProvider" class. This is used in Scheduling of various view windows. It does this using "Schedulerserviceimpl". "Schedulebucket" class allocates one schedule bucket for each EQL statement. Then it adds the Event services using "EventAdapterServices" class. "EventAdapterService" for generating events and handling event types. There are various classes as like "BeanEventAdapter" "BeanEventtype", "PropertyListBuilder" etc. which further sub classify each event so that looking for a particular pattern becomes easier. Also in order for user to import some java packages I have to include "AutoImportServiceImpl" class.Then we had to add some usual database services in the engine for this purpose we used "DatabaseConfigServiceImpl" class.

Then for the services to be embedded in the engine "EPSevicesContext" class is used. First all the services mentioned in the configuration parameter are registered then

other miscellaneous necessary services which are Timer Service, Emit Service (for attaching listeners(those to be notified in case of pattern matching) to the EQL statements), Dispatch Service(For informing the listeners any match of pattern), View Services (for simulating Views just like in case of normal databases), Stream Services(For controlling stream of data like time windows, queues etc), Read Write Lock Services (For modifying views and windows etc at runtime in multithreaded environment), Service for runtime coordination between various services, Admin Services for controlling the engine.

The above is a summarized statement but it actually takes half the amount of code to implements Also when the timer service is added a deamon (background thread) timer is started which notifies Engine of critical events and also used in Callback when any matched pattern is reported by the engine. After doing all the above chores the engine is up for registering for Queries or EQL statements. For registering the query the query is first written as a string and is passed to the engine with the help of "EPAdmin" Class which first passes it to a parser. For parsing purpose a built-in ANTLR Tree Parser is used. For that we have used a standard ANTLR library. Then statement is converted to its Statement Specification using "StatementSpec" Class so that engine can interpret the EQL using the methods of this class. This class has all EQL constructs like Join ,Select , Group etc.. It also provides a managed lock and a View. This View gets a read write lock for runtime modifications. The statement gets a handle in order to manage all its resources like locks, views, bucket etc. Then the statement is invoked to start by providing scheduling bucket. The dispatch service is also registered with the statement. It also gets a pattern stream to control stream of event. Then a listener is added to the "EQLstatement" so that is an event pattern satisfies it the listeners will be notified. Finally the statement is started at the engine and thus finally the engine is up and running with a query. Now we just have to send the events for evaluation.

First the simulator generates random events within the limits of Configuration and then is using the "EPruntimeimpl" class It goes to "EventAdapterService" class which checks for the correctness of Configuration parameters using those registered in "BeanEventType" class and "BeanEventBean". Then its passed to Filter Services for generating patterns. It registers the Callback Service for any matched patterns. The filter service in turn uses various classes and subclasses like "TimeWindowView" "SchedulingServiceView", "OutputProcessView", "EqlView" etc to search for specific pattern. For just a glimpse of complexity for searching a particular pattern we discuss class: public class "ResultSetProcessorFactory":

Here are its properties and job: Factory for output processors. Output processors process the result set of a join or of a view and apply aggregation/grouping, having and some output limiting logic. The instance produced by the factory depends on the presence of aggregation functions in the select list, the presence and nature of the group-by clause.

In case (1) and (2) there are no aggregation functions in the select clause.

Case (3) is without group-by and with aggregation functions and without non-aggregated

properties in the select list: select sum(volume) . Always produces one row for new and old data, aggregates without grouping.

Case (4) is without group-by and with aggregation functions but with non-aggregated properties in the select list: select price, sum(volume). Produces a row for each event, aggregates without grouping.

Case (5) is with group-by and with aggregation functions and all selected properties are grouped-by. in the select list: select customerId, sum(volume) group by customerId. Produces a old and new data row for each group changed, aggregates with grouping, see

Case (6) is with group-by and with aggregation functions and only some selected properties are grouped-by. in the select list: select customerId, supplierId, sum(volume) group by customerId. Produces row for each event, aggregates with grouping.

And here is what we mean by Case(1) or Case(2) etc. Case (1): There is no group-by clause and no aggregate functions with event properties in the select clause and having clause (simplest case) Case (2): A wildcard select-clause has been specified and the group-by is ignored since no aggregation functions are used, and no having clause Case (3): There is no group-by clause and there are aggregate functions with event properties in the select clause (aggregation case) and all event properties are aggregated (all properties are under aggregation functions). Case (4): There is no group-by clause but there are aggregate functions with event properties in the select clause (aggregation case) and not all event properties are aggregated (some properties are not under aggregation functions).

Its job is:

Returns the result set process for the given select expression, group-by clause and-having clause given a set of types describing each stream in the from-clause.

param selectClauseSpec - represents select clause and thus the expression nodes listed in the select, or empty if wildcard

param groupByNodes - represents the expressions to group-by events based on event properties, or empty if no group-by was specified

param optionalHavingNode - represents the having-clause boolean filter criteria

param outputLimitSpec - indicates whether to output all or only the last event

param orderByList - represent the expressions in the order-by clause

param typeService - for information about the streams in the from clause

param insertIntoDesc - descriptor for insert-into clause information

param eventAdapterService - wrapping service for events

param autoImportService - for resolving class names

param viewResourceDelegate - delegates views resource factory to expression resources requirements return result set processor instance

throws ExprValidationException when any of the expressions is invalid

So as there are several Defined EQL terms and statements so there are several classes and interfaces like this. Finally after a pattern is matched the Dispatch Services are called. It in turn calls "UpdateDspatchVew" class which clears the queues and reassigns scheduler etc and then finally informs the listener. Now here also there are various classes and interfaces involved constituting a great deal of code Thus the above is a very brief view of the code that implements our Disang EQL datastream engine.

Chapter 7

Running Example of the engine (Solving the Self Service Terminal Problem)

In this example we consider a self-service terminal system as it exists in airports to allow customers to proceed to self-check in and print boarding passes. The self-service terminal managing system gets a lot of events from all the connected terminals. The event rate is around 500 events per second. Some events indicate abnormal situations such as "paper low" or "terminal out of order." Other events observe activity as a customer uses a terminal to check in and print her boarding pass (see Figure below).

Our primary goal is to resolve self-service terminal or network problems before our customers report them by looking for help, which means higher overall availability and greater customer satisfaction. To accomplish this, we would like to get alerted when certain conditions occur that warrant human intervention: for example, a customer may be in the middle of a check-in process when the terminal detects a hardware problem or when the network goes down. Under these conditions we would like to dispatch a staff member to help that customer, and another staff member to diagnose the hardware or network problem. We also want to provide a dashboard and summarize activity on an ongoing basis and feed this to a real-time interface. This enables a manager to watch the system in action and spot abnormalities. The system can further compare the summarized activity to stored normal usage patterns.

7.1 Events as JavaBeans

Disang is able to handle events as JavaBeans, arbitrary java classes, java.util.Map, or XML documents. In this case study we assume we decided to use the JavaBeans representation for simplicity. Each self-service terminal publishes any of the six events kind below.

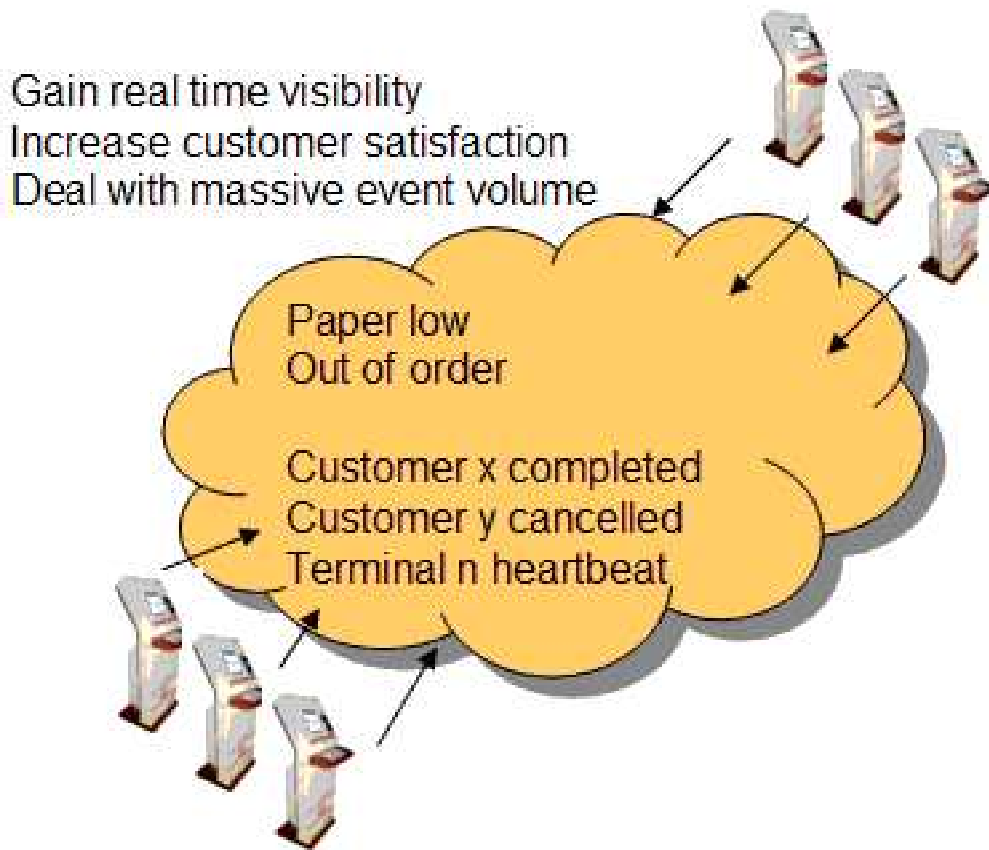

Fig. 7.1 Self service terminal set

Checking	A customer started a check-in dialog
Cancelled	A customer cancelled a check-in dialog
Completed	A customer completed a check-in dialog
OutOfOrder	A terminal detected a hardware problem
LowPaper	A terminal is low on paper
Status	Terminal status, published every 1 minute regardless of activity

Table 7.1 Events of a self-service terminal

All events provide information about the terminal that published the event. Since all events carry similar information, we model each event as a subtype to a base class "BaseTerminalEvent"', which will provide the terminal information that all events issued by a terminal share. A real-world model would of course be more complex– possibly using XML instead.

```
public abstract class BaseTerminalEvent {
    private final Terminal terminal;

    public BaseTerminalEvent(Terminal terminal) {
        this.terminal = terminal;
    }

    public String getType() {
        return this.getClass().getSimpleName();
    }

    public Terminal getTerminal() {
        return terminal;
    }
}
```

For the terminal information we use a simple class to hold the terminal ID:

```
 public class Terminal {
    private String id;

    public Terminal(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }
}
```

The Status event class is thus a subclass of "BaseTerminalEvent":

```
public class Status extends BaseTerminalEvent {
    public Status(Terminal terminal) {
        super(terminal);
    }
}
```

7.2 Introduction to EQL and Patterns

Disang EQL is an object-oriented event stream query language very similar to SQL in its syntax but that significantly differs to be able to deal with sliding window of streams of data. Disang also includes a pattern language that provides for stateful (state-machine) event pattern matching. EQL and patterns can be used alone or can also be combined to express complex temporal logic. Let's assume we want to dispatch staff to restock paper supply when a terminal publishes a LowPaper event. The simple EQL statement below detects such events:

```
select * from LowPaper
```

Besides looking for LowPaper events, we would also like to be notified when OutOfOrder events arrive. We could thus use two separate statements that each filter for only one type of event:

```
select * from LowPaper
select * from OutOfOrder
```

Another solution would be to use a single statement and include an event pattern combined with an or condition:

```
select a,b from pattern [ every a=LowPaper or every b=OutOfOrder]
```

We could also implement this with the help of event polymorphism and the "BaseTerminalEvent". As all events are subclasses of "BaseTerminalEvent", we can use a where clause to filter out the events we are interested in and use the type property (that is the actual class name without package information–see "BaseTerminalEvent"):

```
select * from BaseTerminalEvent
where type = 'LowPaper' or type = 'OutOfOrder'
```

Deciding on a statement merely depends on design choices in this simple case.

7.3 Registering Statements and Listeners

Disang can be configured using either straightforward API, or an XML descriptor. We will use the API here as a Java-centric approach. We then first configure and get an engine instance, register statement(s), and then attach one or more listeners to the created statement(s). The engine allows for nickname to avoid having to specify fully qualified class names in EQL when using JavaBeans event representation:

```
// Configure engine: give nicknames to event classes
Configuration config = new Configuration();
config.addEventTypeAlias("Checkin", Checkin.class);
config.addEventTypeAlias("Cancelled", Cancelled.class);
config.addEventTypeAlias("Completed", Completed.class);
config.addEventTypeAlias("Status", Status.class);
config.addEventTypeAlias("LowPaper", LowPaper.class);
config.addEventTypeAlias("OutOfOrder", OutOfOrder.class);
config.addEventTypeAlias("BaseTerminalEvent", BaseTerminalEvent.class);

// Get engine instance
EPServiceProvider epService =EPServiceProviderManager.getDefaultProvider(config)

// Register statement
String statement = "select * from LowPaper";
EPStatement statement =epService.getEPAdministrator().createEQL(stmt);

// Attach a listener
statement.addListener(new SampleListener());
```

The engine calls all listener classes attached to a statement as soon as new results for a statement are available. Events are encapsulated by an EventBean instance which allows querying the event properties and underlying event class. The engine indicates when events enter a data window via newEvents and leave a data window via oldEvents. Let's look at a sample listener implementation:

```
public class SampleListener implements UpdateListener {
    public void update(EventBean[ ] newEvents, EventBean[ ] oldEvents) {
        LowPaper lowPaper = (LowPaper) newEvents[0].getUnderlying();
        String terminal = (String) newEvents[0].get("terminal");
        String text = (String) newEvents[0].get("text");
    }
}
```

7.4 Detecting the Absence of Status Events

Each self-service terminal publishes a Status event every 1 minute. The Status event indicates the terminal is alive and online. The absence of Status events may indicate that a terminal went offline for some reason and that needs to be investigated. Since Status events arrive in regular intervals of 60 seconds, we can make use of temporal pattern matching using timer to find events that didn't arrive. We can use the every

operator and timer:interval() to repeat an action every 60 seconds. Then we combine this with a not operator to check for absence of Status events. A 65-second interval during which we look for Status events allows 5 seconds to account for a possible delay in transmission or processing:

```
select 'terminal 1 is offline' from pattern
[ every timer:interval(60 sec) ->
(timer:interval(65 sec) and not Status(term.id = 'T1'))]
output first every 5 minutes
```

Since we way not want to see an alert for the same terminal every 1 minute, we added an output first clause to indicate that we only want to be alerted the first time this happens, and then not be alerted for 5 minutes, and then be alerted again if it happens again. See the figure below:

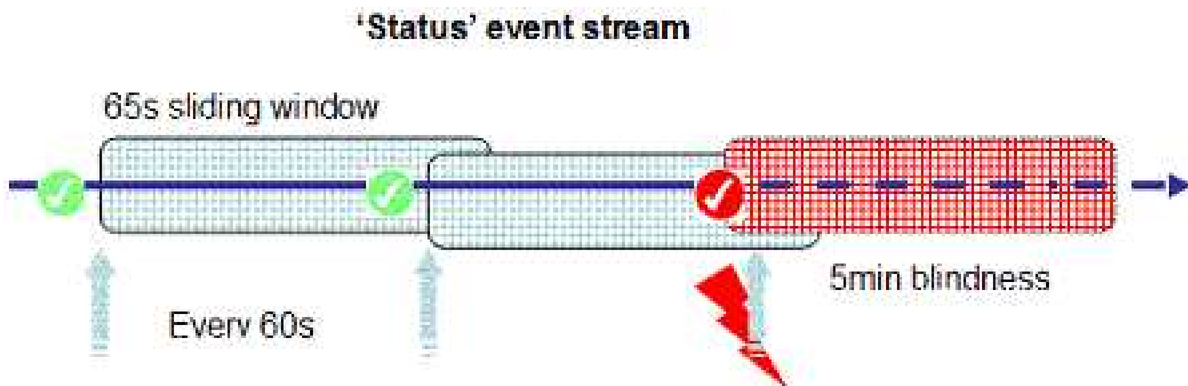

Fig. 7.2 The Status Event Stream (The output first clause can suppress output events)

Note that we hard coded the terminal ID to 'T1' in this query for simplicity. As we are looking for an absent event, it would require to use a subquery to detect all terminal failures with one query.

As you read through this example you probably already realize that Disang ESP/CEP query language expressiveness enables us to do declarative programming in a very loosely-coupled way thanks to the underlying Event Driven Architecture. Most of the detection logic mirroring our business specifications are directly written in statements and not in custom code.

7.5 Activity Summary Data

By presenting statistical information about terminal activity to our staff in real-time we enable them to monitor the system and quickly spot problems. To begin with, the

real-time console should show a count of the number of check-in processes started, in progress, cancelled, and completed within the last 10 minutes. This first query counts the number of Checkin considering only the last 10 minutes of event data:

```
select count(*) from Checkin.win:time(10 minutes)
```

Note the use of the win:time syntax. This tells the engine to consider a time window consisting of only the last 10 minutes of the Checkin event stream.We can improve this query and get a count per event type considering all types of events (Checkin, Completed, Cancelled, Status, OutOfOrder, LowPaper) by using BaseTerminalEvent. Again we want to look at only the last 10 minutes of events so we will use a win:time view. We further want to get notified every 1 minute and not at each change, hence we will use an output all clause:

```
select type, count(*)
from BaseTerminalEvent.win:time(10 minutes)
group by type
output all every 1 minutes
```

Chapter 8

Conclusion and Future Work

So as it is clear from the discussion above that we have been finally successful to create a bare bone structure for a Stream Event Processing Engine. Also the Engine has been created in such a way, using such a language and using such tools that creating a magnificent castle over these barebones will be relatively easier job. There is room for plenty of work like adding additional feature to the engine, such as enabling it Multithreaded sends of events into the engine, Create, start and stop statements during operation, Applications can retain full control over threading, Efficiently sharing resources between statements. Also in future making it such as it can Support multiple independent Disang engines per JavaVM.

Apart from this there are other future innovations possible with this Engine such as making it a Distributed Stream Database Engine such that it enables distributed query processing. Current system doesn't have any library or facility for it. Distributed query means that a query now can spans more than one servers, which implies that now data can be distributed on more than one server. So to produce the result of query, we need to obtain data from both the servers and combine their result.

References

[aur] Brandeis university, brown university, and mit, http://www.cs.brown.edu/research/aurora/. Technical report.

[CN97] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *Proc. of the 1997 Intl. Conf. on Very Large Data Bases*, pages 146–155, 1997.

[DCZ02] M. Cherniack C. Convey S. Lee G. Seidman M. Stonebraker N. Tatbul D. Carney, U. Cetinternel and S. Zdonik. Monitoring streams- a new class of dbms applications. Technical report, Department of Computer Science, Brown University, February 2002.

[DTO92] D. Nichols D. Terry, D. Goldberg and B. Oki. Continuous queries over append-only databases. In *Proc. ACM SIGMOD International Conference on Management of Data.*, pages 321–330, 1992.

[esp] The esper event processing system, http://esper.codehaus.org/index.html. Technical report.

[IP99] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *Proc. of the 1999 Intl. Conf. on Very Large Data Bases*, pages 174–185, 1999.

[NAS96] Y. Matias N. Alon and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of the Annual ACM Symp. on Theory of Computing*, pages 20–29, 1996.

[SAP00] P. B. Gibbons S. Acharya and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Proc. of the Annual ACM Symp. on Theory of Computing*, pages 487–498, May 2000.

[str] The stanford data stream management system, http://www-db.stanford.edu/stream.