Chapter 9
StreamAPAS: Query Language and Data Model

Marcin Gorawski and Aleksander Chrószcz

Summary The system StreamAPAS and its declarative query language allows users to define temporal data analysis. This chapter addresses the problem of lack of the continuous language standard. The proposed language syntax indicates how hierarchical data structures simplify working with spatial data and groups of tuple attributes. The query language is also based on object-oriented programming concepts as a result of which continuous processing applications are easier to develop and maintain. In addition, we discuss the problem of a query logic representation. In contrast to relations stored in DBMS, data streams are temporal so that DSMS should be aware of their dynamic characteristics. Streams characteristics can be described using variables such as tuple rates and invariables like monotonicity. In StreamAPAS, a query is represented as a directed acyclic graph (DAG) whose operators define tuple data transmission model and have information of result stream monotonicity associated with them. Even though this representation is still static, this approach enables us to detect optimization points which are crucial from a stream processing viewpoint.

9.1 Introduction

In the chapter we present the stream processing architecture of StreamAPAS and the prototype query language. There are a lot of research projects which develop a declarative query language for data stream processing (e.g., Babcock et al. 2002; Ali et al. 2005; Yan-Nei et al. 2004). In these researches authors create query languages which syntax bases mainly on the SQL. As a result, users are able to define query upon streams and relations in a convenient way. We should remember

M. Gorawski (✉) · A. Chrószcz
Institute of Computer Science, Silesian University of Technology, Akademicka 16,
44-100 Gliwice, Poland
e-mail: Marcin.Gorawski@polsl.pl

A. Chrószcz
e-mail: Aleksander.Chroszcz@polsl.pl

F. Xhafa et al. (eds.), *Complex Intelligent Systems and Their Applications,*
Springer Optimization and Its Applications 41,
DOI 10.1007/978-1-4419-1636-5_9, © Springer Science+Business Media, LLC 2010

that the SQL is mainly intended for expressing simple queries which are processed over a single database. The data stream queries are different in two aspects. Data Stream Management System (DSMS) usually connects to remote sources and sinks. Besides, the stream processing applications usually need the implementation of custom functions. In our research, we concentrate on the problem of finding the abstract elements that should be introduced to the query language so that the language functionality can be easily adapted to the application requirements.

In defining the semantics and concrete language the following goals were considered:

- Constructing a query language which allows users to define custom functions, import them into the stream processing platform, and use them as native language functions,
- Using a hierarchical data structure, which better suits spatial and analytical data representation,
- Defining a stream processor based on temporal logical operator algebra (Krämer and Seeger 2005), which offers efficient stream-to-stream physical operators.

In contrast to STREAM (Babcock et al. 2002) and ATLAS (Yan-Nei et al. 2004) which are mainly based on the SQL syntax, we added to the query language elements of Object-Oriented (OO) languages. We adopt from OO languages calling class functions, calling object functions, and loading user-defined libraries into the compiler. Thanks to this, the user is able to define new data sources, custom operators, and data sinks by calling library functions from the level of the query language.

The remaining part of this chapter is organized as follows: Sect. 9.2 describes the data stream processing implemented in SreamAPAS; Sect. 9.3 introduces its query language and implementation aspects; next, in Sect. 9.4 we compare our language with CQL language; Sect. 9.5 shows the aims of our further work; and finally Sect. 9.6 presents a summary of our results.

9.2 Data Stream Processing

We use the directed acyclic graph (DAG) to describe the data stream query. DAG nodes represent data stream operators, and edges define stream connections between the operators. We distinguish two levels of a query definition. On the logical level, DAG nodes represent operators of the logic operator algebra (Krämer and Seeger 2005). On the physical level, DAG node describes which algorithm is used to compute a given logical operator, and DAG edge defines how the data communication is implemented. There are a number of data stream processor architectures (Tucker 2005; Krämer and Seeger 2005; Motwani et al. 2003; Abadi et al. 2003). In contrast to them, we develop a stream processor architecture which processes temporal tuples (Krämer and Seeger 2005; Krämer 2007) and positive/negative tuples (Ghanem et al. 2005). Let T be a discrete time domain. Let $I := \{[t_s, t_e) \mid t_s, t_e \in T \wedge t_s \leq t_e\}$ be the set of time intervals.

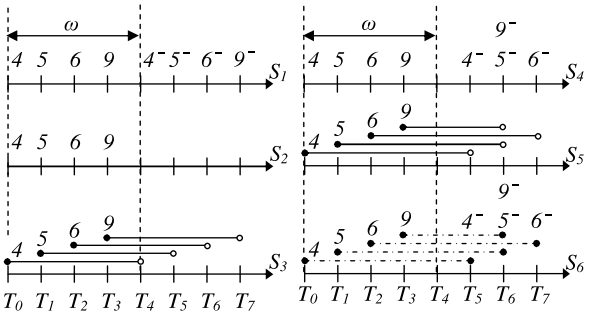

Fig. 9.1 Different stream definition approaches

Definition 9.1 (Stream) A pair $S = (M, \leq_{t_s,t_e})$ is a stream if: M is an infinite sequence of tuples $(type, e, [t_s, t_e])$, where: *type*—tuple type, *e*—attribute tree data, $[t_s, t_e) \in T$. $\leq_{t_s,,t_e}$ is the lexicographical order relation over M (primarily sorting by t_s and secondarily by t_e).

We can calculate the set of valid tuples for a given data stream and a specified point in time t. This set of tuples can be represented in the relational data base as table records which are available in time t. Streams S_1, S_2, and S_3 in Fig. 9.1 illustrate ways of controlling the lifetimes of those records. Stream S_1 uses two tuples so as to define the lifetime of a record. A positive tuple signals the beginning of a record existence, and a negative one points the end. When those lifetimes are constant and equal, we can only transmit the positive tuples as it is shown in stream S_2. Knowing the application time and the lifetime period, we can define a *time window* (Ghanem et al. 2005) which translates the input stream S_2 into the set of valid tuples. When we knew the lifetimes of table records at their time of creation, we can use temporal tuples (Krämer and Seeger 2005) which contain *start* and *end* timestamps as it is shown in S_3. The main advantage of temporal tuples is that they reduce the amount of transmitted data doubly. We cannot achieve this reduction in the model with positive and negative tuples, because their tuples contain only timestamp *start*. The stream S_4 cannot be reduced to positive tuples like S_1 to S_2 because the lifetimes of S_4 tuples are not functions of attribute *start*. However, we can apply the temporal tuples so as to remove negative tuples form the stream as it shows S_5. When we do not know the lifetimes of tuples at their time of creation, the temporal tuple model becomes useless. In some applications, it is acceptable to divide an entry time into the periods. Then, when a period elapses, temporal tuples which represent valid records or events are generated. Unfortunately, this solution generates a lot of additional data, and tuples are one time period delayed. In our research, we develop a system which joins the temporal stream model and the streams with positive and negative tuples.

We use the concepts of *positive temporal tuple* and *negative tuple*. When the lifetime of a table record or an event is known at the time of its generation, it is represented only by *positive temporal tuple*. The lifetimes of those tuples are represented in figures by solid lines. When we do not know the lifetime of the records or events at the time of their generation, we represent them by dashed lines. In such a

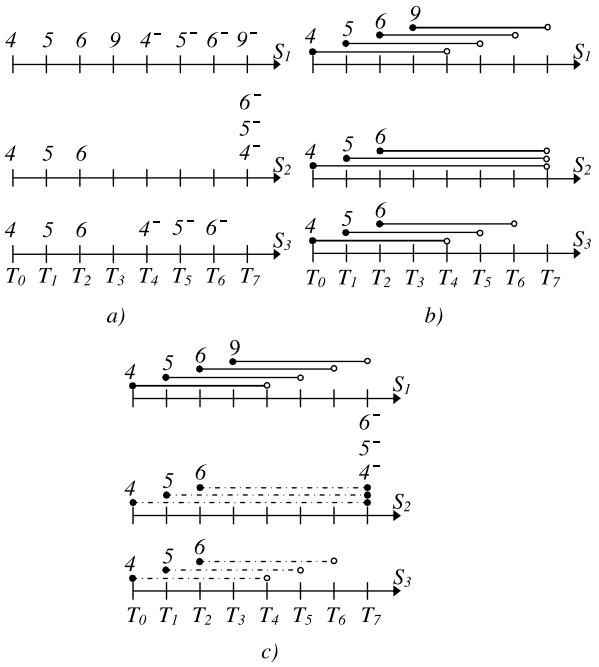

Fig. 9.2 Joining streams in different stream definition approaches

case, the timestamp *end* of a *positive temporal tuple* defines the upper boundary of the tuple lifetime. If we do not know when a tuple can expire, we assign the infinity value to timestamp *end*. When we know that a tuple will expire by a given time, we assign this value to timestamp *end*. The *negative tuple* expires the *positive temporal tuple* in identical way as it is described in model with positive and negative tuples (Ghanem et al. 2005). In comparison with *positive temporal tuple*, the *negative tuple* has zero lifetime period. Those tuples in figures are represented by points. The example of this model is stream S_6 which can substitute S_4.

In Fig. 9.2, we compare join operators for: (a) the model with positive and negative tuples; (b) the temporal model; and (c) the mixed model. Streams S_1 and S_2 are the input streams, and S_3 is the result of a join operator. Stream S_1 transmits only *positive temporal tuples*, and S_2 transmits both types of tuples in Fig. 9.2(c). Let us notice that the result streams S_3 in Fig. 9.2(b) and Fig. 9.2(c) transmit the same tuples, but they have a different interpretation. The result tuples in Fig. 9.2(b) define precisely their lifetimes, whereas the result tuples in Fig. 9.2(c) define the upper boundaries of their lifetimes. Let us note that we have negative tuples in input stream S_2 in example (c), however there is no negative result tuples. This situation happens because the negative tuples arrive at S_2 later than the upper lifetime boundary of result tuples.

In order that physical operators interpret correctly their input streams, each stream has defined monotonicity which is obtained from the operator connected to this stream input. In StreamAPAS, we borrow the stream monotonicity classification from (Golab 2006). Let Q be a query, and τ a point in time. Assume that at τ,

all tuples with lower or equal timestamps have been already processed. The multiset of input tuples at time τ is denoted $S(\tau)$, whereas all the tuples from time 0 to the current time are denoted by $S(0, \tau)$. Furthermore, let $P_S(\tau)$ be the result multiset produced at time τ, and let $E_S(\tau)$ be the multiset of expired tuples at time τ. The equation below defines the result set update function:

$$\forall \tau \quad Q(\tau + 1) = Q(\tau) \cup P_S(\tau + 1) - E_S(\tau + 1).$$

The types of stream monotonicity are defined indirectly. Using the above symbols, we define operators that generate stream of a given monotonicity:

1. The monotonic operator is an operator that produces result tuples which never expire. Formally the property is described by $\forall \tau \forall S E_S(\tau) = 0$.
2. The weakest nonmonotonic operator is an operator that produces result tuples whose lifetime is known and constant. Thanks to that, the order in which those tuples appear at the operator input correspond to the order of their expiration. Formally represented, it looks like $\forall \tau \forall S \exists c \in \mathrm{N} E_S(\tau) = P(\tau - c)$.
3. The weak nonmonotonic operator is an operator whose result tuples have different lifetimes but they are still known at the time of their generation. Let us note that the order of tuple insertion and the order of their expiration are different. This can be formalized as follows: $\forall \tau \forall S \forall S' S(0, \tau) = S'(0, \tau)$, it is true that $\forall t \in P_S(0, \tau) \exists et \in E_S(e) \wedge t \in E'_S(e)$.
4. The strict nonmonotonic operator is an operator whose expiration of tuples depends on the input tuples that will arrive in the future. The lifetimes of tuples are not known at the time of their generation. This can be formalized as follows: $\exists \tau \exists S \exists S' S(0, \tau) = S'(0, \tau)$ and $\exists e \exists t \in P_S(0, \tau)$ such that $t \in E_S(e) \wedge t \notin E'_S(e)$.

The monotonicity of type one says that the tuples of a given stream never expire. This means that the stream of this type transmits only *positive temporal tuples* with infinity assigned to *end* timestamp. The monotonicity of type two is illustrated by S_3 in Fig. 9.1. The stream S_5 in Fig. 9.1 is an example of stream with monotonicity of type three. The last type of monotonicity is illustrated by S_6 in Fig. 9.1. Those examples show that the type of stream monotonicity gives sufficient information to determine a tuple processing algorithm.

From the viewpoint of an operator monotonicity, there are two management types of result streams:

- The direct approach in which an operator calculates the lifetime of a result tuple directly at the time of the tuple generation. The tuple's lifetime is determined only via its timestamp *start* and *end*. Hence, the expired tuples can be determined using the application time without the need for negative tuples. Here we classify the operators defined over *time-based* windows.
- The negative tuple approach exists when an operator is assigned to the operator that generates a negative tuple or the operator is defined over a count type window (such as a *slide window*). As a consequence, an operator result stream has negative tuples. This management type has two disadvantages. The output streams of the operators have nearly twice as many tuples in comparison to result streams in

the direct approach. Moreover, operator *count type window* uses more memory resources.

Let us notice that the higher number of stream monotonicity, the more complicated architecture of the tuple collection which is connected to a given stream. Stream monotonicity of numbers: 1, 2, and 3 process no *negative tuples*. Those collections check only timestamp *end* so as to find expired tuples. If a stream is a weakest nonmonotonic one, the tuples expiration order is identical to the stream order. As a result, potentially expired tuples exist only at the beginning of tuple collection. In consequence, a simple list data structure is enough to implement this tuple collection. The stream monotonicity of type three has two potential implementations. The expired tuples can be identified by testing all the elements of a collection, or we can add an additional list in timestamp *end* order.

Suppose that there exists a query which consists of a few join operators. Because the join operators are commutative, we can change the order of their processing in a query plan. When we reduce the number of operators with high number of monotonicity type in a query plan production, then we also reduce the number of more complicated and slower tuple collections. When we put an operator which generates the negative tuple on a higher position in a query production plan, the lower number of operators became strict nonmonotonic. When we put the operators fed by the weakest nonmonotonic operators at lower position in a query production plan, we reduce the number of tuple collections that process the weak nonmonotonic streams. The above rules are added to a rule optimizer which reorders query operators in the following ways:

- Selection operators are shifted to the lowest acceptable positions in a query,
- Window operators are lifted to the highest acceptable positions in a query.

The created nearly-optimizer is aware of operator monotonicity; it reorders query operators in such a way that a query production plan has less complicated and slower tuple collections. In comparison to the nearly-optimizers based on statistics such as stream rates and operator selectivity, our optimizer identifies the complexity of data collection management.

It is worth noticing that the operator monotonicity, which is a static operator property, enriches the description of a stream query (DAG) in such a way that the introduced nearly-optimizer is able to identify sub-DAGs which are of benefit to further query optimization. Next, those sub-DAGs can be dynamically optimized with the help of two algorithms, adaptive caching (Babu et al. 2005) and data synopses (Arasu et al. 2006). Another area of further optimizer research is creation of composed operators which consist of basic physical operators. Suppose that our optimizer identifies a group of strictly nonmonotonic operators; then this knowledge can be used to create a composed operator which shares the tuple collections between physical operators in order not to duplicate nonmonotonic collections.

9.3 Query Language

Many query languages have been proposed to stream databases such as CQL (Arasu et al. 2006), Cayuga (Demers et al. 2007), Esper, and Streaming SQL (Namit et al. 2008; Yijian et al. 2006) which belong to declarative languages. Even though the Object Representation of Query (ORQ) is a commonly used part of SDMS, we have not come across extensions of query languages which use elements of the Object-Oriented (OO) paradigm in order to automatize mapping new DSMS functionalities to the query language. The development of most stream query languages can be named descending, because at the beginning a new syntax of language extension is defined, and then it is implemented in ORQ. In contrast to them, we used elements of OO paradigm in order to invert the development of our query language. Thanks to that, we can extend the object representation of a query, and it is automatically available form the query language level. Moreover, this approach systematizes the way the language evolves. Let us notice that the OO paradigm can also make the query language syntax more confusing when the object representation of a query is complicated.

Now we will specify the *data factories* and the *data collections* which are used in our language presentation. *Data collections* describe the schemes of streams or relations. Moreover, they are accessed by *data factories*. The OO paradigm is used to represent *data factories* as objects which supply methods that transform a data factory into a stream or a relation.

Let us follow the example below. We want to create a tuple-based window on stream S with the size of five tuples. According to our notation, this is expressed by `S{rangeWindow(5)}`, where `S{...}` indicates the *data factory* related to stream S. `rangeWindow(5)` is an object method which creates the tuple-based window with the size of five tuples.

9.3.1 Structure of Query Language

The StreamAPAS users define *units* which represent groups of queries and then use commands `compile`, `run` and `remove` to control the state of those *units* in DSMS. This approach simplifies the management of query resources, because it introduces a higher level of abstraction where we do not have to control each subquery individually. Stream processing applications are defined as an acyclic directed graph whose nodes represent operators and edges represent data transfer. Those operators are created and configured by means of *tasks* which define the production plans of streams (and other data structure in future). The query language syntax allows us to define *tasks* directly by means of methods and indirectly by a syntax similar to SQL/CQL.

StreamAPAS is implemented in Java, and therefore we have decided to allow users to extend the functionality of DSMS by means of packages.

We have defined the following syntax of method call:

```
[<fully qualified class name>.]<method name>
            [::method modifier]([<arg list>])
```

where:

- a fully qualified class name is a class name with a path to the name scope from which the class is referenced. This notation is equivalent to Java notation of fully qualified class name.
- a method modifier is an element which modifies the way how a method call is interpreted. Currently the system defines *task* modifier.

When we call a method which is defined in a current local scope, then we do not indicate a *fully qualified class name*. We have to write a *fully qualified class name* when we call a static method (class method). The *task* modifier is used when a method creates an object *task*. It is necessary for the compiler to arrange the hierarchy of task name scopes correctly.

Example 9.1 We want to collect the times of result latencies measured for the *distinct* operator.

```
test run
  begin
    Benchmark.RandomStream::task("I")
    S{Set.distinct(I{}, "valL")}
    Gui.showAndRegisterLatency::task(S{},"out.txt")
end;
```

This example is solved by a *unit* named `test`. It consists of a *task* which generates random stream I. Then stream S is defined which is a result of the `distinct` operator. Next, S is visualized, and the result latencies are saved to file "out.txt." Let us notice that *task* objects are created by the class methods `RandomStream` and `showAndRegisterLatency`. The object representing the operator `distinct` in ORQ is also created by class method `Set.distinct`. Figure 9.3 shows the hierarchy of *units* and *tasks* for Example 9.1.

The queries below show the full syntax of *unit*. Additionally, the language offers a shortcut which allows users to define a *unit* and run it in one call as it was described in the previous example.

```
//compilation and registration
test compile
  begin

  end;

//starting query
Test run;

//removing query
test delete;
```

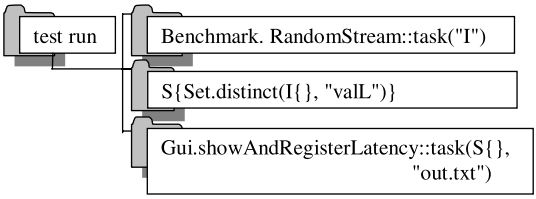

Fig. 9.3 The query division into unit and tasks for Example 9.1

Usually the data processing system consists of a number of streams which cooperate only at the level of a given *unit* or *task*; so this is not necessary to make them accessible from different levels. Therefore *units* and *tasks* create a tree in which each node has an associated scope of *data factory* names.

Figure 9.3 illustrates an example structure of name scopes. We have implemented the following management of name scopes. Let us assume that we refer to data factory I by the query language. This means that the compiler searches for I declaration, at first, in the current local scope and then recursively in the parent scopes. This mechanism is enriched with shortcut objects which reside in a local name scope and point at *data factories* defined in other name scopes. Those shortcuts can be used to refer to *data factories* declared in higher levels of name scope hierarchy by single name.

The hierarchy of name scopes needs each data factory to be a qualified name which consists of a data factory name and the path which leads from a *unit* node to the name scope where the data factory is defined. In consequence, each name scope node has to be uniquely identified by name. Let us notice that users do not have to use qualified names directly to refer to *data factories* declared in children scopes because those *data factories* are reachable through shortcuts. Thanks to that, the compiler can automatically assign unique names to *task* nodes, whereas the name of the *unit* node is specified by user.

In order to create shortcuts, we have defined the *public* modificator to *data factories*. This modificator orders the compiler for the creation of a shortcut to a *given data factory* into the parent scope. In Example 9.1, modifier *public* is assigned to stream I. Thanks to that, this stream is reachable from other tasks inside the *unit*.

9.3.2 Syntax of Unit and Task

Data stream databases are the subject of intensive research which covers new schedulers, stream operators, indexing structures, and DSMS architectures. Therefore we have decided to develop a language which requires little effort to extend it in order to test those new functionalities.

Adding new functionalities to the SQL language is connected with reediting the language syntax. Our aim is to reduce the necessity of language syntax changes in

such situations. In consequence, it will be easier to adapt it to the changing DSMS environment.

Each *unit* and *task* can be mapped to an object which belongs to ORQ. Let us notice that those objects can be created and manipulated directly by the query language if only a query language supports the OO paradigm. In consequence, changes of ORQ would be automatically mapped onto the query language. In order to implement this, each element in the language has its own name scope which contains the names of object methods. In the current system, we distinguish three areas which have defined object methods. The name scope of a *unit*'s methods which is a body of *units* between `start` and `end`. The name scopes of the *data factories* and the attribute tree nodes which are delimited by `{}` and `[]`. The implementation details of how object methods are made accessible from the query language are discussed in Sect. 9.3.4.

Example 9.2 We want a *unit* to use scheduler `XYZ`. This scheduler is defined in the `scheduler` package, and the method which configures `XYZ` is named `BasicCfg`.

```
test run
  begin
    setScheduler(shedulers.XYZ.BasicCfg())
    ...
  end;
```

In the example above, there is a class method called `BasicCfg` which creates an object that represents the configuration of scheduler `XYZ`. Then this configuration is passed to the *unit* by calling object method `setScheduler`.

Summing up, this example illustrates the power of expressiveness when we allow users to create some parts of ORQ directly from the query language. Let us notice that the language syntax is static, and only the contents of the name scope are subject to change.

9.3.3 Attribute Tree

Stream data bases can be classified as data warehouses which are intended to calculate nearly real response time. In such applications, the data organized hierarchically facilitates the manipulation and interpretation of results. Hierarchical data can be used to group attributes thematically. For instance, a car can be described by a unique identifier and a node which represent its position. Then the position node can consist of attributes x, y. Hierarchical data is also useful in reflecting the organization of aggregates. SQL and CQL languages define relation or stream schemas as a list of attributes. When we want to create a data schema similar to a hierarchical data structure by means of SQL, it is necessary to define new custom data types.

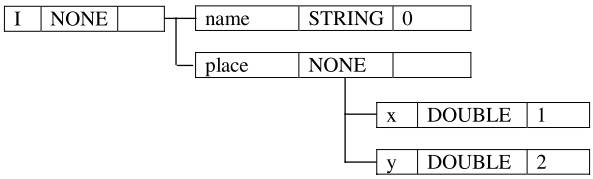

Fig. 9.4 Sample attributes tree

Contrary to the SQL approach, the attribute tree allows us to create a hierarchical data structure as a composition of simple expressions.

The attribute tree is represented by nested lists of nodes which have names and declarations of the node value type. These declarations consist of object type declaration and index value indicating the localization of object values in tuples. The label of the attribute-tree root represents the name of the data collection (e.g., stream). Not all the nodes contain values; those empty nodes are of `none` type. Such nodes are useful in defining groups of attributes. For example, the *place* node in Fig. 9.4 represents the group of position attributes.

If we define a query expression which reads attribute `I.place.x` (Fig. 9.4), the compiler gets the *index* attribute from the `I.place.x` node, and then this *index* = *1* value is used to get the attribute values form the tuples.

The basic syntax of tree logic formulas is presented below. It results from syntax proposed in Cardelli and Ghelli (2001).

η ::= label expression
  \$*name* search for a node in namespace of data collections
  name search for a node in current namespace
A, B ::= formula
  true subtree of current node
  $\eta[A]$ location of current node
  A, B composition of formulas

The composition of formulas allows us to create a new data structure from primitive formulas. The query language is able to check the equivalence of attribute tree structure types, thanks to that subtrees of attribute trees and single nodes can be used as function operands. Suppose that we want to use the subtree of the `place` node as an operand. In this situation we write the name `$I.place.x[true]`.To simplify the syntax, e.g., `O{z[x = 1.1]}`, the series of brackets `[]` can be replaced by `O{z.x = 1.1,...}`.This syntax is known in OO languages as "dot syntax." At the time of the declaration of an attribute-tree node, the node has no data type. The first assignment operation that is called on this node determines its type. When we assign a value to an attribute-tree node that has a predefined data type, an attempt is made to cast the value type on the data type of this node. If there is no defined cast for a given data type, the compilation error will be sent. When we refer to the data collection object, we have to use braces `{}`. This syntax allows the compiler to distinguish whether we operate on tuples or on data collections.

In the example below, we show how to declare the stream which consists of the sum of place coordinates and the subtree of the `place` node.

```
O{z = $I.place.x + $I.place.y,
  $I.place[true]}
```

At the beginning of this formula, we declare `O` stream. Then we declare the stream as a composition of attributes `z` and `place`. Next, we define how the values of those attributes are calculated. The formula not only defines the attribute tree but also the *data collection* object.

The use of `$` depends on whether a fully qualified name is needed. At the beginning, the name scope points to a catalog of available data collections, and therefore the name of data collection is not preceded by `$`.When we are inside the declaration of a data collection, the other collections and their attributes must be fully qualified.

Summing up, when the data collection schemas are well designed, long lists of attributes which are met in SQL can be replaced with a shorter one consisting of attribute tree nodes. The language which is defined to manipulate attribute trees joins two functionalities. This enables the user to define the *data collection* schema and calculation plans together.

9.3.4 Functions

The limited number of available data types diminishes any system usability. To avoid this, we have implemented abstract types in the StreamAPAS query language. When a new data type is needed, we have to add new custom functions that create and operate this new object type. For instance, the `RandomStream` function creates object that implements the `task` interface.

Our compiler has been implemented in Java, so we have decided to use a reflective programming paradigm. In order to add a new function to the query language, the user defines this function in a java class. Then *classpath* to this class has to be added to the library manager of the compiler. The reflection allows us to search methods by their names and their argument lists inside *.class* file. Let us notice that arithmetic functions are called when streams are processed, whereas the `RandomStream` function is called during the compilation phase. Therefore, the compiler needs additional information on the role of the method. We associate this information with method by means of the annotation mechanism. In default, all methods are recognized as arithmetic functions, when a given method has to be called during the compilation phase, we annotated method with `@MModificator(mode = MModificator.CUSTOM_OP_BASE)`. This mechanism allows us to integrate specialized function modifiers defined in the query language with the Java language code.

Reflection gives unlimited access to all the methods defined inside classes. This access policy is unacceptable because some methods should be achievable only from the Java code not from the query language. In order to define a new access policy,

we use the annotation mechanism. In default, all the methods are visible to the query language. When we want to hide all the methods of a given class, we annotate the class with `@CModificator(mode = CModificator.HIDEMEMBERS)`. When some method should be accessible to the query language, the role in the query language have to be assigned directly.

Summing up, it is not a complicated task to define a new custom function and add it to the query language because we need the syntax of Java language only. Moreover, this approach simplifies testing new methods. The java code bellow shows sample declaration of the `StreamUniformRandom` class function.

```
@CModificator(
 mode = CModificator.HIDEMEMBERS)
public class benchmark {

@MModificator(
 mode=MModificator.CUSTOM_OP_BASE...)
public static OperatorBase
                                StreamUniformRandom(...
    ...
```

Because data collections in the query language are also represented as objects, a data collection can have methods that are accessible at the query language level. Similarly to class functions, the object functions are annotated by `@MModificator( mode = MModificator.CUSTOM_OP_BASE)`. An example of object function is `rangeWindow`.

9.4 Linear Road Benchmark

The simplified version of the original linear road benchmark will be used to compare syntaxes of another stream query language named CQL with our language.

The linear road application computes the fee for each vehicle on the motorway individually, in such a way that vehicles visiting congested segments of the motorway pay a higher fee. In consequence, the traffic on the motorway is balanced, because drivers chose other routes so as to minimize the fee to be paid. This benchmark assumes that each vehicle has a sensor which transmits the values of speed and location. Then this information is transferred through a sensor system to the central server which updates the rates and the fees. Next, the updated individual fee and rates for using the motorway segments are forwarded back to vehicle's sensors. A detailed description of the linear road benchmark is available in Arasu et al. (2004).

Figure 9.5 describes the elements of the motorway system. There are L lanes which are numbered $0, \ldots, L-1$. Each lane is 100 miles long and runs east–west. The motorway is divided into 100 segments whose boundaries have entrance and exit ramps. Vehicles on the motorway transmit their positions and speeds every

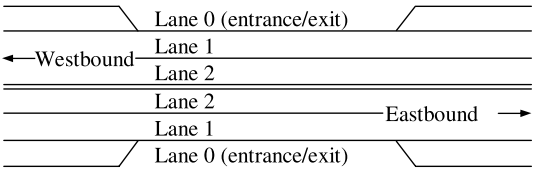

Fig. 9.5 A sample segment of the Linear Road motorway

30 seconds. The position is defined as a lane number, a direction (east/west), and a distance measured from the left end of the motorway.

Vehicles pay a fee when they go through congested segments. A segment is considered congested when the average speed of all the vehicles in this segment during the last 5 minutes is lower than 40 MPH. The fee rates are calculated according to the following formula: $baseFeeRate * (numVehicule - 150)^2$.

In the remaining part of the chapter, we will assume that the linear road application generates stream *SegSpeedStr*(*vehiculeId, speed, segNo, dir, hwy*). The attribute *vehiculeId* identifies a vehicle, *speed* is the speed in MPH, *segNo* denotes segment where vehicle is, *dir* denotes the direction (east/west), and *hwy* denotes the motorway number.

9.4.1 CQL

CQL originates from SQL which was extended by two language syntaxes that correspond to stream-to-relation and relation-to-stream operators. In consequence, queries in CQL are easy to express by users familiar with SQL. In order to illustrate the CQL syntax, we will show examples of queries which resolve parts of the liner road benchmark.

Example 9.3 We want to know which vehicles are active. A vehicle is active when it has transmitted its position during the last 30 seconds.

```
Select Istream(distinct vehiculeId)
From SegSpeedStr[Range 30 Seconds]
```

The above query illustrates the usage of all the operator classes. First, a stream-to-relation operator represented by the time-sliding window is applied. Then, the output relation is processed by two relation-to-relation operators, projection and duplicate elimination, respectively. Finally, the result relation is converted into a stream with a relation-to-stream operator represented by `Istream`.

In order to reduce common operators in expressions, CQL specifies the following syntactic shortcuts. If a query omits the specification of a stream-to-relation operator and the semantic needs a relation, then the compiler applies window `S[Range Unbounded]`. The compiler also inserts `Istream` after the root operator of Q when the result of Q is monotonic and a relation-to-stream operator is not specified. Those rules applied to the previous example result in the following query:

```
Select distinct vehiculeId
From SegSpeedStr[Range 30 Seconds]
```

Other similarities between CQL (Arasu et al. 2006) and SQL are illustrated by the queries below.

Example 9.4 We want to create a relation which contains all the segments containing active vehicles.

```
Select distinct L.vehiculeId, L.segNo, L.dir, L.hwy
From SegSpeedStr[Range 30 Seconds] as A,
     SegSeedStr[Partition by vehiculeId Row 1] as L
Where A.vehiculeId = L.vehiculeId
```

Example 9.5 We want to create a relation which contains all the congested segments.

```
Select segNo, dir, hwy
From SegSpeedStr[Range 5 Minutes]
Group By segNo, dir, hwy
Having Avg(speed) < 40
```

Example 9.6 We want to calculate the number of vehicles in segments.

```
Select segNo, dir, hwy,
       count(vehiculeId) as numVehicles
From ActiveVehiculeSegRel
Group by segNo, dir, hwy
```

9.4.2 Example Queries in StreamAPAS

The query bellow shows how Example 9.3 can be expressed in our language.

```
select result{$SegSpeedStr.vehiculeId}
where SegSpeedStr{slideWindow(30 000)}

resultDist{Set.distinct(result{}, "vehiculeId")}
```

At first, a time-based window is declared with the size of 30 seconds. Then the result is saved to stream *result*. In the next sub query, a `distinct` operator is defined which saves *result* to stream *resultDist*. It is worth noticing that the `distinct` operator is created by calling the class method: `Set.distinct`. In contrast to that, the time-based window is created with the use of the object method `slideWindow`. This method is declared in the name scope of the *data factory* which represents stream *SegSpeedStr*.

The following queries show how Examples 9.4–9.6 defined in CQL can be expressed in our language.

Example 9.7

```
Select tmp{$L.vehiculeId, $L.segNo, $L.dir, $L.hwy}
From L{SegSpeedStr{}}
Where SegSpeedStr{slideWindow(30 000)},
            L{partitionedWindow(1,"vehiculeId"),
            SegSpeedStr.vehiculeId = L.vehiculeId

ActiveVehicleSegRel{Set.distinct(tmp{},
                                  "vehiculeId")}
```

Example 9.8

```
select CongestedSegRel{$SegSpeedStr.segNo.segNo,
                       $SegSpeedStr.segNo.dir,
                       $SegSpeedStr.segNo.hwy}
where SegSpeedStr{slideWindow(300 000)}
group by SegSpeedStr.segNo, SegSpeedStr.dir,
         SegSpeedStr.hwy
having Agg.sum($SegSpeedStr.speed) < 40
```

Example 9.9

```
Select SegVolRel($ActiveVehiculeSegRel.segNo,
                 $ActiveVehiculeSegRel.dir,
                 $ActiveVehiculeSegRel.hwy,
                 numVehicles = Agg.count())
group by ActiveVehiculeSegRel.segNo,
         ActiveVehiculeSegRel.dir,
         ActiveVehiculeSegRel.hwy
```

9.5 Further Work

Development of data warehouses in streaming environments is a promising area of new applications, and this constitutes the basis of our further work. In the chapter we have illustrated the query language. Its syntax is motivated by our previous tests with indexing structures designed for spatio-temporal data.

Let us notice that any indexing structure can be easily represented as a new *task*. This process requires us to load package with new functionalities. Then we can use its class methods to define the specification of an indexing structure, in a way similar to the one shown in Example 9.2 and the examples below.

Example 9.10 We want to create index *traffic* which is an R-tree supplied with streaming information of vehicle positions transmitted by stream *I*. The query bellow shows the potential beginning of this solution.

```
gis.Rtree::task("traffic", {$I.point[true]},...)
```

Example 9.11 Then we want to use index *traffic* to find the nearest five vehicles which may be sent to accidents. Let those accidents be notified by stream *help* which transfers tuples with accident positions. The query below shows the *data factory* as an interface used to define operations on the index structure.

```
select result{$traffic{}}
where traffic{kNN($help{})};
```

Example 9.12 The syntax of the query language associates a tree with a *data factory*. This tree represents dimensions and subdimensions. Besides, each node has its own scope of function names. Thanks to that we can look at defining queries upon indexes in a way similar to the one used in data warehouses. In order to define the operator which extracts data from a *data factory*, the user calls the methods of this node tree. Let us assume that index *traffic* is an aggregate tree and we want to calculate the number of vehicles falling into a given area. The example below illustrates how it could be expressed with the use of the *data factory* syntax.

```
select result{$traffic{}}
where traffic{contain ($areas{}), measures[sum()]};
```

9.6 Related Work

There are a lot of stream processing research projects such as STREAM (Motwani et al. 2003), Telegraph (Shah et al. 2001), TelegraphCQ (Sirish et al. 2003), and Aurora & Borealis (Balazinska 2006). There are as many stream query languages proposals as many projects are carried. We can divide those languages into three categories. In one approach, each operator connection is defined directly in a text file or graphically (Balazinska 2006). In a procedural language, a user defines query as a loops that are fed with tuples from stream collections. The most promising role plays declarative languages (Motwani et al. 2003) because their users may be not aware of the physical realization (e.g., chosen algorithms). There, the query optimizer is responsible for translating a task defined on an abstract level into a physical realization.

Stream processing becomes popular in online analysis based on aggregates defined over different windows, data sequence, prediction, and many more. Undoubtedly, those functionalities are closer to data warehouse tools rather than traditional database operations. Therefore, we not only consider SQL syntax as a basis for further research but also CQL and MDX gain our interest. There are a lot of propositions of SQL syntax extensions which usually address a particular class of problems, for instance, time sequence analysis (Ali et al. 2005).

9.7 Conclusion

In the developed language, we test the combination of the Object-Oriented paradigm and declarative languages including the languages designed for multidimensional data analysis. The proposed query language is not a complete solution; however, it has a systematized approach to functionality extensions so that the implementation of the syntactic analyzer is easier.

In the chapter, we describe the impact of query logic representation on an optimization phase. The proposed query nearly optimizer reduces the number of strict nonmonotonic operators as a result of which the system transmits the lower number of negative tuples. Moreover, we show how the information on monotonicity of operator can be used to accelerate tuple collections governed by physical operators.

Finally, we introduce the attribute tree which represents spatial and analytical data in a more convenient way, because it allows the user to define expressions on single attribute or a group of attributes.

The current stream processing engine supports only stream data collections. It is a subject of our further research to add relations and more sophisticated index structures into the system.

References

Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The VLDB Journal **12**(2), 120–139 (2003)

Ali, M.H., Aref, W.G., Bose, R., Elmagarmid, A.K., Helal, A., Kamel, I., Mokbel, M.F.: NILE-PDT: a phenomenon detection and tracking framework for data stream management systems. In: VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases, pp. 1295–1298. VLDB Endowment (2005)

Arasu, A., Cherniack, M., Galvez, E.F., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: A stream data management benchmark. In: M.A. Nascimento, M.T. Özsu, D. Kossmann, R.J. Miller, J.A. Blakeley, K.B. Schiefer (eds.) VLDB, pp. 480–491. Morgan Kaufmann, San Mateo (2004)

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

Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS '02: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 1–16. ACM, New York (2002)

Babu, S., Munagala, K., Widom, J., Motwani, R.: Adaptive caching for continuous queries. In: ICDE '05: Proceedings of the 21st International Conference on Data Engineering, pp. 118–129. IEEE Computer Society, Washington (2005)

Balazinska, M.: Fault-tolerance and load management in a distributed stream processing system. Ph.D. thesis, Cambridge, MA, USA (2006)

Cardelli, L., Ghelli, G.: A query language based on the ambient logic. In: ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems, pp. 1–22. Springer, London (2001)

Demers, A.J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W.M.: Cayuga: A general purpose event monitoring system. In: CIDR, pp. 412–422 (2007)

Ghanem, T.M., Hammad, M.A., Mokbel, M.F., Aref, W.G., Elmagarmid, A.K.: Query processing using negative tuples in stream query engines. Tech. Rep. 04-040, Purdue University (2005)

Golab, L.: Sliding window query processing over data streams. Ph.D. thesis, University of Waterloo (2006)

Krämer, J.: Continuous queries over data streams semantics and implementation. Ph.D. thesis, Philipps-Universität Marburg (2007)

Krämer, J., Seeger, B.: A temporal foundation for continuous queries over data streams. In: COMAD, pp. 70–82 (2005)

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query processing, resource management, and approximation in a data stream management system. In: CIDR, pp. 245–256. CIDR (2003)

Namit, J., Shailendra, M., Anand, S., Johannes, G., Jennifer, W., Hari, B., Çetintemel, U., Mitch, C., Richard, T., Stan, Z.: Towards a Streaming SQL Standard. pp. 1379–1390. VLDB Endowment (2008)

Shah, M.A., Franklin, M.J., Madden, S., Hellerstein, J.M.: Java support for data-intensive systems: experiences building the telegraph dataflow system. SIGMOD Record **30**(4), 103–114 (2001)

Sirish, C., Owen, C., Amol, D., Wei, H., Sailesh, K., Samuel, M., Vijayshankar, R., Frederick, R.: TelegraphCQ: Continuous dataflow processing for an uncertain world. In: CIDR (2003)

Tucker: Punctuated data streams. Ph.D. thesis, OGI School of Science & Technology At Oregon Heath (2005)

Yan-Nei, L., Haixun, W., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: Proceedings of the VLDB International Conference of Very Large Data Bases, pp. 492–503 (2004)

Yijian, B., Hetal, T., Haixun, W., Chang, L., Zaniolo, C.: A data stream language and system designed for power and extensibility. In: CIKM '06: Proceedings of the 15th ACM International Conference on Information and Knowledge Management, pp. 337–346. ACM, New York (2006)