Dynamic Metadata Management for Scalable Stream Processing Systems

Michael Cammert, Jürgen Krämer, Bernhard Seeger
Department of Mathematics and Computer Science,
University of Marburg, Germany,
{cammert,kraemerj,seeger}@informatik.uni-marburg.de

Abstract

Adaptive query processing is of utmost importance for the scalability of data stream processing systems due to the long-running queries and fluctuating stream characteristics. An essential prerequisite for adaptive runtime components is the presence of suitable metadata capturing the runtime state. As most of the metadata in such a system gets outdated over time, appropriate update mechanisms are required. Dynamic metadata management *deals with the dynamic provision and continuous maintenance of metadata. This paper does not only address the issues in dynamic metadata management such as metadata dependencies and metadata update concepts, but also presents a scalable framework to efficiently manage the diversity of dynamic metadata in todays data stream processing systems. The core of our field-tested metadata framework is a publish-subscribe architecture that enables the system to identify and compute only the currently required metadata. This tailored metadata provision is crucial to scalability as maintaining all available metadata at runtime causes significant computational overhead when the number of continuous queries increases.*

1 Introduction

Any system that processes data requires metadata because it represents important background information on the data itself and also on queries over the data. The use of metadata is well-known from conventional database systems. For example, schema information is a form of metadata describing the data format, while cardinality and selectivity estimations are metadata mainly used for query optimization purposes. Although query processing in scalable stream processing systems (SSPS) differs from that in conventional database systems, SSPS require metadata as well because queries are long-running and query execution has to be highly adaptive as query workload and stream characteristics are expected to change over time. For that reason, we raised the question whether SSPS need novel types of metadata and novel techniques for metadata management to deal successfully with stream processing and the arising adaptivity and scalability issues. To the best of our knowledge, this question has not been addressed in the literature so far and, hence, is the starting point for this paper.

A SSPS is designed to run thousands of continuous queries concurrently. In order to enable subquery sharing, query execution is based on a large graph composed of operators. Metadata may refer to the *sources* of the query graph, i. e., the raw data streams, or they belong to the *operators* inside the graph, or they refer to the *sinks*, i. e., entire queries (see Figure 1). As the term *metadata* is typically used in the plural, we denote the single parts making up the metadata as *metadata items*. Metadata items on sources are stream rates, data distributions, schema information, etc. Operator metadata items are selectivities, resource usage, implementation type (nested-loops, hash-based), and so on. Finally, metadata items at query-level are, for instance, Quality-of-Service (QoS) specifications, (scheduling) priority, or frequency of reuse by subquery sharing.

We distinguish between two basic classes of metadata: *static* and *dynamic* metadata (see Figure 2). Static metadata are invariable, whereas dynamic metadata are likely to change at runtime. Examples for static metadata items are general stream information like schema or element size. However, most of the metadata in a SSPS belongs to the latter class, namely dynamic metadata. Stream rates, operator resource usage, and selectivities are examples for dynamic metadata items. A SSPS profits from monitoring dynamic metadata as it enables the system to adapt its behavior during query execution. Without adequate runtime statistics obtained from dynamic metadata, a data stream management system cannot react appropriately to fluctuating stream characteristics and changing query workload, and thus is not scalable. Totally different system components can conduct those runtime adaptations to improve scalability, e. g., scheduler, memory manager, or query optimizer. Here are some motivating applications that make extensive use of dynamic metadata:

1. **Scheduling.** The Chain scheduling strategy [5] has to react to significant changes in operator selectivities to minimize the memory usage of inter-operator queues.

2. **Resource Management.** Metadata on resource allocation is necessary to apply load shedding techniques [21] with the aim to keep overall resource usage in bounds.

3. **Query Optimization.** Changes in stream characteristics, such as stream rates or value distributions, may necessitate re-optimizations at runtime, e. g., a left-deep join tree is migrated to its right-deep counterpart [25, 18].

4. **System Profiling.** Researchers and administrators may also benefit from runtime metadata because its analysis gives insight into system behavior. Metadata profiling is often useful for system configuration or experimental performance evaluations.

Up to this point we only motivated that a SSPS needs a variety of metadata. However, all these individual metadata items have to be provided by the system itself. But providing all available metadata would be too expensive. The reason is that the majority of metadata in a SSPS is time-varying and consequently needs to be updated in a continuous manner. The more metadata items need to be updated, the higher the update costs. As operators in a query graph provide metadata, a larger query graph leads to increased metadata update costs. For scalability reasons, it is thus not satisfactory to compute all available metadata. Another important criterion for scalability is the tradeoff between metadata freshness and computational overhead. A higher update frequency causes higher computational costs. The continuous metadata update process prevents to apply the classical techniques known from database systems which compute the metadata once prior to query execution. Therefore, new approaches to metadata management have to be developed to meet the requirements of the highly dynamic environment given in a SSPS.

In this paper, we propose a *publish-subscribe architecture* for metadata management. This architecture enables a SSPS to provide metadata on demand, i. e., only the metadata items that are actually required will be supplied and maintained. In addition, it allows sharing common metadata items. Although such a solution seems to be straightforward at first glance, we want to show that things get complex very fast when having a closer look. In this paper, we address two major issues:

- **Metadata dependencies.** A metadata item may depend on several other metadata items. For example, an estimation of the memory usage of a sliding window join depends on the window sizes and the input stream rates. Those dependencies need to be resolved automatically. Hence, a system has to support the definition of metadata dependencies. In general, this means to maintain a dependency graph at runtime.

- **Metadata maintenance.** Different types of metadata necessitate different update methods. Some metadata items are updated on access, while others are updated periodically. Moreover, metadata dependencies have to be considered. As a consequence, metadata items have to be updated whenever a metadata item they depend on is refreshed. For instance, the average input rate depends on the input rate. If the input rate is measured at runtime and updated periodically, the average input rate should be updated in addition.

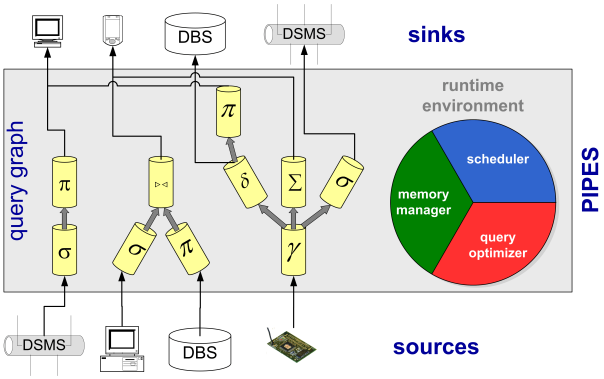

Figure 1. Overview of the PIPES stream processing infrastructure

To summarize, *dynamic metadata management* deals with the dynamic provision and the continuous maintenance of metadata. The contributions of this paper are as follows:

- We present our publish-subscribe architecture for dynamic metadata management. This architecture is designed to cope with metadata dependencies. According to the metadata dependencies defined, dependent metadata items are automatically included and excluded.

- We describe our metadata maintenance concepts. We show which problems may occur if metadata items need to be updated, and provide solutions based on different update mechanisms: on-demand, periodic, or triggered update. In addition, we reveal the tradeoff between metadata freshness and computational overhead.

- As this paper summarizes our experiences with realizing dynamic metadata management in our stream processing framework PIPES [17, 8], we finally spent some words on implementation issues.

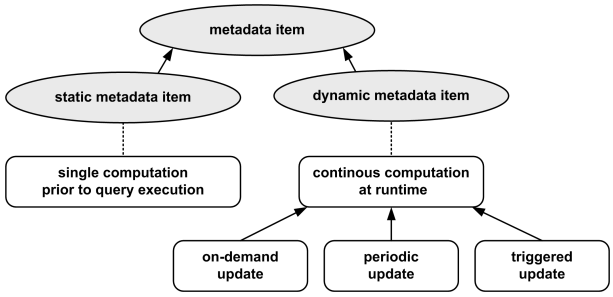

Figure 2. Metadata types and maintenance concepts

The remainder of this paper is organized as follows. Section 2 shows up metadata dependencies and motivates our publish-subscribe approach. Our update mechanisms for time-varying metadata are discussed in Section 3. Section 4 outlines important implementation issues. Related work is discussed in Section 5 and Section 6 summarizes the paper.

2 Publish-Subscribe Architecture

A *metadata consumer* needs to access metadata. Metadata consumers can be system components, operators, users, etc. The set of metadata items required in a SSPS at runtime is defined by the metadata consumers. Note that this set is likely to vary over time, e. g., when new queries are installed. This necessitates a flexible architecture that allows managing the metadata items provided at runtime. As maintaining dynamic metadata at runtime is expensive, only the metadata actually needed should be maintained to ensure scalability. Moreover, shared access should be granted for metadata items because multiple consumers may require the same metadata item. The previous reasons inspired us to choose a *publish-subscribe architecture* as an appropriate and scalable basis for dynamic metadata management.

2.1 Metadata Access

In our publish-subscribe architecture, consumers have to subscribe to the metadata items they need. An incoming subscription causes the system to create and return a so-called *metadata handler*. There is a 1-to-1 relationship between metadata items and metadata handlers. A metadata handler can be considered as a proxy that supplies the subscribed metadata consumers with the current metadata value. This indirection is required because (i) it synchronizes the possibly concurrent access of multiple consumers, and (ii) it guarantees a consistent view on a metadata item for all consumers during updates. For the case that a handler already exists for the requested metadata item, the subscription returns the existing handler and increments a counter for this item. Thus, sharing handlers saves redundant maintenance costs. An unsubscription to a metadata item decrements the corresponding subscription counter, and causes the subsequent removal of the associated handler if the counter reached zero. Since a lot of metadata handlers are updated frequently, the automated removal of handlers, which are no longer needed, saves further system resources.

2.2 Metadata Storage

In our metadata management architecture, metadata items and handlers are assigned to graph nodes. Recall that the query graph represents the continuous queries executed in a SSPS (see Figure 1). In our understanding, a query graph consists of sources at the bottom providing the data in form of raw data streams. The intermediate nodes are operators processing the data streams, whereas the sinks at the top establish the connections to the applications by providing query results. The metadata items and handlers are stored at the respective graph nodes, e. g., the selectivity of a join is directly stored at the join operator node in the query graph, and QoS specifications provided by an application are stored at the corresponding sink. This direct assignment of metadata to the individual graph nodes facilitates metadata discovery because each node gives information about available metadata items.

2.3 Metadata Dependencies

In many cases, complex metadata items can be computed from other metadata items. For example, the input/output ratio of an operator can be derived from dividing the input rate by the output rate. In that case, the input/output ratio depends on two metadata items, namely the input and output rate. If the system already provides both metadata items, the new metadata item can be defined and computed efficiently. Note that in our architecture, developers can define and register metadata dependencies. Those definitions are the basis for the automatic subscription and unsubscription of dependent metadata items. In the case that metadata management architectures do not permit the reuse of existing metadata, support for new metadata items would have to be implemented from scratch. This would not only be time-consuming for the developers integrating new metadata items into the systems but also expensive to maintain as underlying information may be stored and updated in a redundant manner.

Note that metadata dependencies are not restricted to a single node. A metadata item may depend on metadata items from several other nodes. Therefore, we distinguish between the following two types of metadata dependencies:

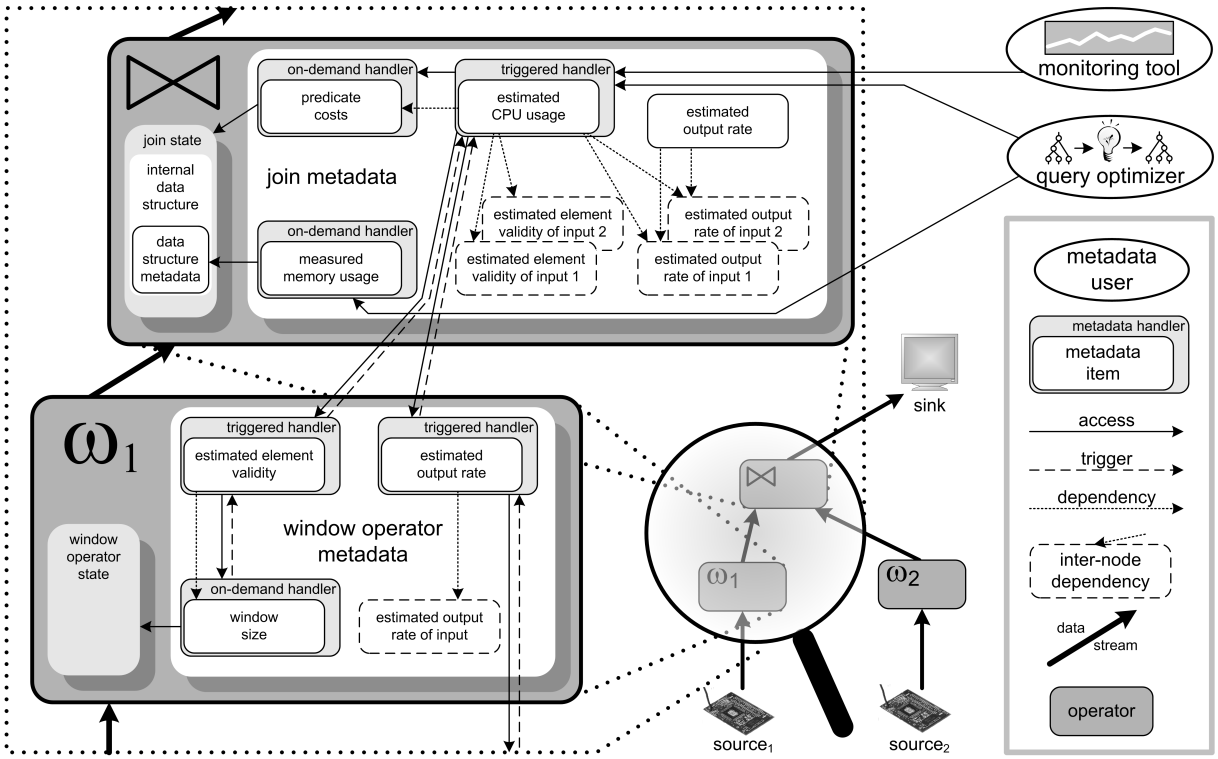

Figure 3. Dynamic metadata management for a time-based sliding window join

- **Intra-Node Dependencies.** A metadata item may depend on metadata items supplied by the same node. Online aggregates of local metadata items belong to this type, e. g., the average or variance of the join selectivity.
- **Inter-Node Dependencies.** A metadata item may depend on metadata items supplied by other nodes. For example, an estimation of the memory usage of a sliding-window join needs to know the estimated stream rates of its two inputs as underlying metadata which are provided at the operators upstream. But also metadata items from nodes downstream may be required, e. g., if an operator needs to access QoS specifications.

2.4 Automated Metadata Inclusion

If the responsibility for resolving metadata dependencies is left to the consumers, they would have to include/exclude all dependent metadata items manually by explicit subscriptions/unsubscriptions. Be aware that metadata dependencies can be very complex as dependent metadata items may, in turn, depend on other items. To overcome the laborious process of manually resolving those dependencies, we decided that our architecture manages all metadata dependencies in a directed graph. This graph is used for an automatic inclusion and exclusion of dependent metadata items. Whenever a consumer subscribes to the metadata item of interest, a depth-first traversal of the dependency graph is performed starting at the requested metadata item. All traversed items are implicitly included and supplied henceforth. The traversal stops at items already provided. For an unsubscription, the same traversal cancels the provision of dependent metadata items by an implicit exclusion.

2.5 Example

A significant part of the findings in this paper are results from our experiences with integrating a suitable and powerful cost model for runtime resource management in PIPES. In order to illustrate the complexity and functionality of our publish-subscribe architecture, Figure 3 sketches the estimation of the CPU usage of a time-based sliding window join over two data streams. The query plan for this continuous query consists of the two data sources, followed by the window operators (ω), the join ($\bowtie$), and a sink that consumes the results. As we do not want to delve into the details of query semantics and plan construction here, we only want to mention that windowing constructs are usually implemented by a separate operator in SSPS, namely the window operator. In the case of a time-based sliding window, this operator assigns a validity to each incoming stream el-

ement according to the window size. Suppose a monitoring tool should plot the estimated CPU usage of the join, maybe with the aim to compare it with the currently measured CPU usage. Then, the monitoring tool subscribes to the metadata item *estimated CPU usage*. As a consequence, a metadata handler is created which ensures the computation and update of this item. Note that an item without a handler indicates that this item is available but unused, e. g., the estimated output rate of the join in Figure 3. Inter-node dependencies are used in our example to include metadata items that provide metadata on stream rates and element validities from the inputs of the join. As the expected output rate of a window operator depends on the expected output rate of its input, this example also shows that dependencies may proceed recursively. The costs of the join predicate can be obtained using an intra-node dependency.

3 Dynamic Metadata Maintenance

Contrary to static metadata, dynamic metadata should be kept up-to-date in order to capture temporal changes in the metadata. The maintenance mechanisms in our dynamic metadata management approach address two important objectives: (i) the overhead for maintaining metadata should be kept low, and (ii) multiple accesses to the same metadata item must not influence each other. The latter aspect can be considered as an isolation condition.

3.1 Motivation

Let us first motivate the need for different update mechanisms. Besides static metadata, some metadata items can be maintained by an *on-demand* evaluation. That means, the value of such an item is solely updated when the item is accessed. For example, the measured memory usage of an operator results from the sizes of its internal data structures implementing the operator state multiplied with the sizes of the stream elements. In the case of binary window-join, two data structures store the elements in the windows, one data structure for each input.

For other metadata items, an on-demand update is not satisfactory and can lead to inconsistent results in the worst case. Let us consider the measurement of the input rate of an operator. An on-demand update would mean that the input rate is computed for the point in system time at which the corresponding item is accessed. In more detail, the input rate could be computed by counting the number of elements that arrived since the last access divided by the time period between the current and the last access. However, this kind of update causes problems if two metadata consumers access the same item nearly at the same time. While the first consumer will get the correct value, the value returned to the second one will not be meaningful because the element counter has been reset shortly before and the time period passed will be very small. Hence, the value returned to the second consumer will often be zero. For that reason, we introduce a second update type, namely *periodic* updates, which compute metadata values for a fixed time window. Such a value is considered as valid during the fixed time period defined by the window. The window size is a parameter in our approach that allows calibrating the tradeoff between freshness and computational overhead. Note that the metadata value is not updated for each incoming element, but each element is still considered in the result as the overhead for counting incoming elements is low. This is an important aspect for scalability because if metadata items were updated for each element, the computational overhead for metadata management in a data stream system would be too high compared to the processing costs.

Because the value of certain metadata items can only be outdated if one of its underlying metadata items has been changed, a periodic update would waste resources. In order to improve scalability, we introduced a third type of updates called *triggered* update. Based on our metadata dependency concept, this update mechanism allows updating metadata values whenever it is necessary, i. e., one of the underlying items was updated. For example, refreshing the metadata value for the measured input rate triggers the update of the value for the measured average input rate due to the dependency between the input rate and its average.

3.2 Update Mechanisms

In the following, we describe the three update mechanisms motivated above (see also Figure 2). In addition, we discuss how updates proceed across nodes. Recall that the *metadata handler* of a metadata item is responsible for updating the metadata value.

3.2.1 On-Demand Updates

Our first update mechanism is the *on-demand* update. This intuitive type of update is applicable in many common cases. The metadata handler computes the new metadata value on demand whenever the corresponding metadata item is accessed. With respect to our design goals, this mechanism can be used for

- metadata items that are probable to be accessed rarely. An on-demand computation is the cost-optimal solution in that case, whereas pre-computing those metadata values would obviously waste system resources.
- metadata items that are cheap to compute. In many cases, a metadata item can be supplied (i) by forwarding already existing information in a node, e. g., information on the operator state or internal data structures,

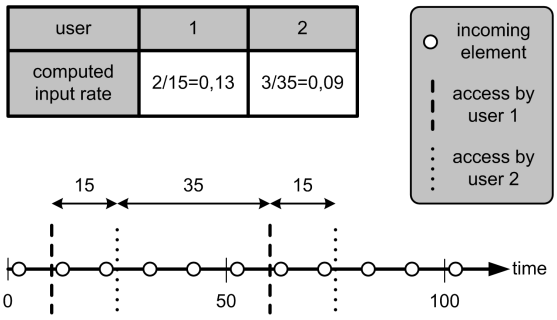

Figure 4. Problems with concurrent periodic access

or (ii) by applying simple computations to the existing information, which can be other metadata items. In both cases, the overhead of pre-computing and updating the metadata value would be higher than the costs of its computation on access.

- metadata items that essentially depend on freshness. If a metadata consumer needs the exact value of a metadata item at access time, an on-demand computation is the sole, suitable solution as it ensures highest accuracy.

3.2.2 Periodic Updates

The second update mechanism is the *periodic* update that refreshes a metadata value regularly based on a fixed time window. During each time period determined by the window size, information is gathered. At the end of a time period the new metadata value is computed based on that information. Finally, the existing metadata value is replaced with the new one. This metadata value is returned by the corresponding handler on each access during the following time period. One could argue that the returned value is outdated by at most one period in the worst case. However, this mechanism guarantees our isolation condition claimed above, namely that an asynchronous or even concurrent access of multiple metadata consumers should return a consistent and correct metadata value (see also our motivating example in Section 3.1).

Figure 4 shows a scenario where two users want to compute the same metadata value, namely the input rate, concurrently. The time period between two subsequent accesses of either user is 50 time units. The element arrival rate is constant. Although all involved events – element arrival and metadata access – occur in a periodic manner, the metadata computations of both users interfere with each other. While the correct input rate is obviously 0.1, both users compute incorrect rates (see table at the top of Figure 4). The input rate is computed by the division of the number of incoming elements since the last access and the time passed since then. If only a single metadata handler periodically computes the input rate, it always reports the correct value. In our framework, therefore, the users retrieve the correct metadata value from the corresponding handler.

3.2.3 Triggered Updates

Our third update mechanism is the *triggered* update. Similar to the second one, a triggered metadata handler always returns a pre-computed metadata value when accessed. In contrast, the corresponding metadata value is not refreshed periodically but on an event. Metadata dependencies are used to determine which metadata handlers have to be triggered. Whenever the value of a metadata item changes that is maintained by a periodic or triggered handler, all dependent triggered handlers are notified and updated automatically. Note that those updates can trigger updates on dependent metadata handlers in turn, and so on. As a consequence, triggering updates may proceed recursively following the edges of the inverted dependency graph.

Because metadata values of on-demand handlers are only updated on access, other dependent metadata items are therefore not instantly notified of changes. If changes of such metadata items should be reflected immediately, e. g., changes in the operator state, we additionally allow to define a notification for this event. Whenever the event occurs, the notification is passed to all dependent metadata handlers and triggers the necessary updates. Hence, the definition of event notifications enables the developer to fire triggers manually. This mechanism guarantees that the pre-computed metadata values of triggered handlers are correct and up-to-date if a triggered handler depends on an on-demand handler.

Having the objectives (i) overhead minimization and (ii) isolated metadata access in mind, triggered updates are used for:

- **Update propagation.** The values of metadata items with triggered handlers are pre-computed on the first subscription. They are only updated on certain predefined events or when an underlying metadata item changes. This causes fewer costs than a periodic update to ensure metadata freshness. Updates of metadata values are not limited to a single node because a metadata handler can be triggered according to an inter-node dependency. Updates can therefore propagate through the query graph.

  To propagate updates across nodes in a query graph, a node retains information about other nodes that initiated a metadata inclusion. Based on this information, a node can determine all connected nodes expecting notifications about metadata changes. Duplicate subscriptions by the same node are detected to

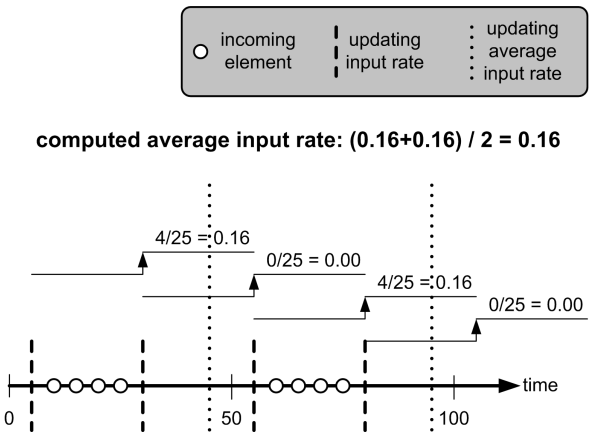

Figure 5. Problems with on-demand aggregation

avoid redundant notifications. After a connected node cancelled all subscriptions on a metadata item, it will not be notified about changes concerning this item any more.

- **Synchronization.** Often it might be intuitive to define an on-demand handler for a metadata item that depends on an item with a periodic handler. For example, the input rate is updated periodically, while the average input rate could be computed on access by an online average. Although this solution sounds convincing, it causes serious problems if the access to the handler returning the average value is not synchronized with the update of the input rate. As a consequence, (i) incorrect metadata values may be returned because updates of the input rate have not been considered in the average if the frequency of updates to the input rate is higher than the access frequency of the average input rate, or (ii) an unwanted weighting is performed if the access frequency of the average input rate is higher than the update frequency of the input rate.

  Figure 5 gives an example for case (i). While the updates on the input rate correctly cover the bursty nature of the element arrival, the less frequent updates on the average input rate are always computed for the peak input rate, which results in a wrong average value.

In order to provide correct and consistent metadata values, two aspects have to be considered in general: (i) updates have to be performed in the right order, and (ii) updates need to be synchronized. The update order is basically determined by the inverted dependency graph. The problem shown in the above example can be solved by replacing the on-demand handler with a triggered handler. Due to the dependency, an update of the input rate triggers the re-computation of the average input rate.

3.3 Example

Let us consider our cost model example from Section 2.5 again. In [9] we proposed an approach to adaptive resource management for sliding window queries that relies on adjustments to window sizes at runtime. Whenever the window size is changed by the resource manager, the cost estimations for the operator resource usage have to be updated according to our cost model. As shown in Figure 3, the estimation of the join CPU usage depends on the estimated element validity which, in turn, depends on the window size. Moreover, the CPU usage of the join depends on the output rate of its inputs. For the dependent items, we have chosen triggered handlers. When the window size is changed, an event is fired. This event triggers the handler of the estimated element validity due to the intra-node dependency between the estimated element validity and the window size. An inter-node update triggers the re-estimation of the join CPU usage whenever the estimated element validity or the estimated output rate of one of the join's inputs was updated.

4 Implementation

The dynamic metadata management framework presented in this paper is fully implemented as an integral part of the stream processing infrastructure PIPES [17, 8]. PIPES is not only made for programmers who want to set up a scalable stream processing system, but also supports data stream research due to its extensible and highly generic design, which allows to implement and evaluate new techniques easily. While the metadata functionality of PIPES has already been harnessed in several demonstrations [17, 8], the underlying metadata management framework addressed in this paper has neither been illustrated nor published before. During our implementation work we had to deal with the following issues.

4.1 Intuitive Usage

It should be easy for users of PIPES to access the manifold metadata available. This design goal is accomplished by the publish-subscribe architecture described in Section 2. The user just has to subscribe to the metadata items of interest. It does not matter whether a metadata item is already in use or not. Furthermore, the user does not need to know how a data item is computed. This includes the knowledge about metadata dependencies.

4.2 Synchronization Problems

We had to take care of synchronization problems due to the multi-threaded query processing engine of PIPES. This not only includes the concurrent access of multiple metadata consumers, but also the concurrency between the processing of stream elements and metadata access. For example, the state of a join has to be updated for each incoming element, while metadata items referring to the state can be accessed at the same time. The locking mechanism in PIPES consists of three different types of reentrant read-write locks controlling access at graph-, operator-, and metadata level. Further technical details are omitted due to space constraints.

4.3 Scalability and Efficiency

The metadata management framework should be scalable and efficient. The publish-subscribe mechanism guarantees that only the metadata items actually required are provided at runtime. This saves system resources since unused metadata items are not maintained. With regard to multi-threading, only the locks involved in the computation of the currently included metadata items are used to guarantee isolation. A further optimization for scalability is to distribute the periodic update tasks over a small pool of worker-threads. For small query graphs, however, a single thread is sufficient to handle all periodic updates.

4.4 Flexible and Extensible Architecture

The development of a flexible and extensible architecture for metadata management was one of our design goals [7]. As the nodes and runtime components in PIPES are highly generic, the metadata management framework also has to be very flexible with the aim to cover the different types of metadata required in a SSPS.

4.4.1 New Metadata

It should be intuitive for developers to enrich nodes with new metadata. Prior to the definition of a new metadata item, a developer has to (i) identify the required information to compute the metadata value, (ii) determine existing metadata items that provide parts of this underlying information, and (iii) choose an appropriate update handler. The implementation consists of the following three steps.

- Metadata dependencies have to be defined. This includes local dependencies as well as dependencies to nodes upstream and downstream. Usually, those dependencies are static and can be expressed easily by calling the pre-defined dependency definition functions of our framework.

- The *addMetadata* method constructs the metadata handler for each metadata item. To provide a new metadata item, the developer has to extend this method by registering a new metadata handler. PIPES provides pre-implementations of metadata handlers for the update mechanisms described in Section 3.2 to facilitate the implementation work. The developer just has to parameterize them with a function that evaluates the metadata value. This function can involve (i) any locally available information, e. g., state information, and (ii) all metadata items for which a dependency has been defined. Some metadata items require the node to gather additional information. For example, the input rate requires to count the number of incoming elements. The developer has to add specific *monitoring code* for that purpose which needs to be activated by the *addMetaData* method. The *addMetadata* method is called by the publish-subscribe mechanisms when a metadata item is included for the first time.

- The *removeMetadata* method removes a metadata handler, disables the computation of the metadata value, and deactivates the execution of the corresponding monitoring code. Hence, it is the reverse operation to *addMetadata* and typically easy to implement.

4.4.2 Metadata Inheritance

When a developer extends the class implementing a node in order to add specific functionality, he/she inherits all the metadata provided by the super class. While defining new metadata items can be achieved as described above, the developer also has the option to override the definition of existing metadata items. Let us consider an example of an operator that provides a metadata item for its memory usage. If a specialized implementation speeds up the operator by using additional data structures, the allocated memory for the additional data structures has to be reflected in the memory usage metadata item. Furthermore, it might be necessary to overwrite existing dependencies for similar reasons. Our metadata framework is designed to support the redefinition of metadata items and dependencies.

4.4.3 Dynamic Dependencies

For complex operators and other dynamic metadata environments the static dependency resolution proposed for our metadata framework may not be sufficient. Therefore, PIPES also offers to override the pre-implemented resolution mechanism. In particular, this enables the developer to dynamically redefine dependencies. Consider for example a metadata item A computable from a metadata item B. Consequently, a dependency is defined pointing from A to B.

Assume, item A can alternatively be computed from metadata item C. If item C has already been included at runtime, but B has not, the dependency for A can be redefined such that A points to C. This saves computational resources because the unnecessary inclusion of B is prevented when A is included.

4.5 Metadata of Exchangeable Modules

Due to the generic design of PIPES, many operators depend on exchangeable modules, e. g., the join operator can be based on different data structures to store its state (lists, hash tables, etc.). Metadata items can also depend on properties of these modules. For that reason, we extended our metadata framework towards these modules. As a result, metadata items defined for an operator can access metadata items specified in the operator's modules. For instance, the memory usage of the join relies on the memory usage of the internal data structures as shown in Figure 3. The metadata framework is applied recursively to access metadata items of nested modules.

5 Related Work

Metadata plays an important role in traditional database management systems (DBMS) [23]. In this context, the term metadata mostly refers to information about relations in a database. This involves information about schemas, indices, cardinality estimations, data distributions etc. Standard DBMS are not well-suited for processing data streams due to the high degree of adaptivity required for processing the long-running queries over fluctuating data streams [14]. On the one hand, some metadata items can be used analogously to DBMS, e. g., static metadata required by the query optimizer for plan construction and static query optimization. On the other hand, adaptive query processing mainly requires dynamic metadata (see Section 1). Todays DBMS are neither designed to make use of time-varying metadata during query processing nor have mechanisms for dynamic metadata management.

In the area of distributed databases, peer-to-peer databases, and web services, metadata management is also a relevant topic. However, metadata issues in those areas differ from the ones presented in this paper as they predominantly focus on the discovery of appropriate data sources according to a given query [20]. A distributed approach to metadata management is proposed in [16]. Based on a publish-subscribe architecture, users and applications specify the information they need, which is provided locally through caching and replication. This work is complementary to ours as it (i) is a distributed approach, and (ii) considers static metadata describing Internet resources. Our work is aimed at adaptive query processing in a SSPS and mainly addresses the management of time-varying metadata.

In recent years several prototypes for data stream systems have been developed, e. g., [19, 1, 11, 17]. As motivated in Section 1, all these systems need to access metadata in order to be adaptive. Metadata required for scheduling are utilized in [5, 10, 15]. Metadata to adaptively control system resources like memory usage are addressed in [6, 9, 2]. Query optimization has attracted a lot of research attention [22, 25, 24, 3, 4, 12]. But any query optimization needs runtime statistics as a form of metadata. Stream processing systems often apply load shedding to keep resource usage in bounds. To determine the point in time when load shedding techniques should be performed as well as their extent necessitates the presence of appropriate runtime statistics [13, 21]. Overall, metadata are a key factor in any stream processing system. However, to the best of our knowledge, we are the first that face the issues in dynamic metadata management.

6 Conclusion

SSPS require a diversity of metadata, especially runtime metadata, for adaptivity and scalability. Despite the large body of work on stream processing, the issues in metadata management have not been addressed so far. The initial part of the paper motivated the general need for metadata in todays stream processing systems and resulted in a categorization of different metadata types and their update requirements. Then, we proposed a novel publish-subscribe architecture to efficiently deal with the various types of metadata, tailored metadata provision, and automatic dependency resolution. The different maintenance concepts available in our metadata framework ensure metadata freshness and avoid computational overhead, an important prerequisite for scalability. Throughout the paper, we revealed subtleties of dynamic metadata management obtained from our experiences in developing the field-tested SSPS infrastructure PIPES. Furthermore, we discussed design goals and implementation issues ranging from usability over extensibility to scalability. We believe that our work answers some important questions about dynamic metadata management, and hope that future stream processing systems can profit from our insights and solutions.

Acknowledgments

This work has been supported by the German Research Foundation (DFG) under grant no. SE 553/4-3.

References

[1] D. J. Abadi and D. Carney et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

[2] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *TODS*, 29(1):162–194, 2004.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of ACM SIGMOD*, pages 261–272, 2000.

[4] A. Ayad and J. F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *Proc. of ACM SIGMOD*, pages 419–430, 2004.

[5] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proc. of ACM SIGMOD*, pages 253–264, 2003.

[6] S. Babu, U. Srivastava, and J. Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *TODS*, 29(3):545–580, 2004.

[7] J. Bercken, B. Blohsfeld, and J.-P. Dittrich et al. XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. In *Proc. of VLDB*, pages 39–48, 2001.

[8] M. Cammert, C. Heinz, J. Krämer, T. Riemenschneider, M. Schwarzkopf, B. Seeger, and A. Zeiss. Stream Processing in Production-to-Business Software. In *Proc. of ICDE*, pages 168–169, 2006.

[9] M. Cammert, J. Krämer, B. Seeger, and S. Vaupel. An Approach to Adaptive Memory Management in Data Stream Systems. In *Proc. of ICDE*, pages 137–139, 2006.

[10] D. Carney, U. Cetintemel, S. Zdonik, A. Rasin, M. Cerniak, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *Proc. of VLDB*, pages 838–849, 2003.

[11] S. Chandrasekaran, O. Cooper, and A. Deshpande et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of CIDR*, 2003.

[12] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proc. of ICDE*, 2002.

[13] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive load shedding for windowed stream joins. In *Proc. of CIKM*, pages 171–178, 2005.

[14] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5–14, 2003.

[15] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, 2003.

[16] M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. A Publish & Subscribe Architecture for Distributed Metadata Management. In *Proc. of ICDE*, pages 309–320, 2002.

[17] J. Krämer and B. Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proc. of ACM SIGMOD*, pages 925–926, 2004.

[18] J. Krämer, Y. Yang, M. Cammert, B. Seeger, and D. Papadias. Dynamic Plan Migration for Snapshot-Equivalent Continuous Queries in Data Stream Systems. In *EDBT Workshops*, pages 497–516, 2006.

[19] R. Motwani, J. Widom, and A. Arasu et al. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *Proc. of CIDR*, 2003.

[20] G. Singh, S. Bharathi, and A. Chervenak et al. A Metadata Catalog Service for Data Intensive Applications. In *ACM/IEEE Conf. on Supercomputing*, page 33, 2003.

[21] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *Proc. of VLDB*, pages 309–320, 2003.

[22] S. D. Viglas and J. F. Naughton. Rate-based Query Optimization for Streaming Information Sources. In *Proc. of ACM SIGMOD*, pages 37–48, 2002.

[23] R. Williams, D. Daniels, L. Haas, G. Lapis, L. P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. R*: An overview of the architecture. In *Readings in database systems*, pages 196–218, 1988.

[24] Y. Yang, J. Krämer, D. Papadias, and B. Seeger. HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. *TKDE*, 2007. (to appear).

[25] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *Proc. of ACM SIGMOD*, pages 431–442, 2004.