Quality-of-Service-Aware Data Stream Processing

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Dipl.-Inf. Sven Schmidt
geboren am 6. März 1978 in Zwickau

Gutachter: Prof. Dr.-Ing. habil. Wolfgang Lehner
Technische Universität Dresden, Fakultät Informatik
Institut für Systemarchitektur, Lehrstuhl für Datenbanken
01062 Dresden

Prof. Dr.-Ing. habil. Klaus Kabitzsch
Technische Universität Dresden, Fakultät Informatik
Institut für angewandte Informatik, Lehrstuhl für Technische Informationssysteme
01062 Dresden

Prof. Dr. Bernhard Seeger
Philipps-Universität Marburg, FB Mathematik und Informatik
AG Datenbanksysteme
Hans-Meerwein-Straße
35032 Marburg

Tag der Verteidigung: 13. März 2007

Dresden im Oktober 2006

Abstract/Kurzfassung

Quality-of-Service-Aware Data Stream Processing

Data stream processing in the industrial as well as in the academic field has gained more and more importance during the last years. Consider the monitoring of industrial processes as an example. There, sensors are mounted to gather lots of data within a short time range. Storing and post-processing these data may occasionally be useless or even impossible. On the one hand, only a small part of the monitored data is relevant. To efficiently use the storage capacity, only a preselection of the data should be considered. On the other hand, it may occur that the volume of incoming data is generally too high to be stored in time or–in other words–the technical efforts for storing the data in time would be out of scale.

Processing data streams in the context of this thesis means to apply database operations to the stream in an on-the-fly manner (without explicitly storing the data). The challenges for this task lie in the limited amount of resources while data streams are potentially infinite. Furthermore, data stream processing must be fast and the results have to be disseminated as soon as possible.

This thesis focuses on the latter issue. The goal is to provide a so-called Quality-of-Service (QoS) for the data stream processing task. Therefore, adequate QoS metrics like maximum *output delay* or minimum *result data rate* are defined. Thereafter, a cost model for obtaining the required processing resources from the specified QoS is presented. On that basis, the stream processing operations are scheduled. Depending on the required QoS and on the available resources, the weight can be shifted among the individual resources and QoS metrics, respectively.

Calculating and scheduling resources requires a lot of expert knowledge regarding the characteristics of the stream operations and regarding the incoming data streams. Often, this knowledge is based on experience and thus, a revision of the resource calculation and reservation becomes necessary from time to time. This leads to occasional interruptions of the continuous data stream processing, of the delivery of the result, and thus, of the negotiated Quality-of-Service. The proposed robustness concept supports the user and facilitates a decrease in the number of interruptions by providing more resources.

Datenstromverarbeitung unter Beachtung von Qualitätsanforderungen

Die Verarbeitung von Datenströmen erlangte in den letzten Jahren sowohl im akademischen als auch im industriellen Umfeld immer mehr Aufmerksamkeit. Zum Beispiel entstehen bei der Überwachung industrieller Prozesse durch geeignete Sensoren eine enorme Menge von Daten in kürzester Zeit. Eine Aufzeichnung und nachträgliche Auswertung ist nicht immer sinnvoll und vor allem mitunter nicht möglich. Zum einen sind nicht alle Daten relevant und es ist wichtig, eine geeignete Vorauswahl zu treffen, um Speicherplatz effizient zu nutzen. Zum anderen können – je nach Umfang der Datenerfassung – so viele Daten erzeugt werden, dass eine zeitgerechte Speicherung mit enormem technischen Aufwand verbunden wäre.

Datenstromverarbeitung in diesem Kontext bedeutet, die Daten ohne explizite Speicherung ('on-the-fly') durch Operationen angelehnt an die klassische Datenbanktechnologie zu verarbeiten. Die Herausforderungen dabei sind die nur begrenzt zur Verfügung stehenden Ressourcen bei potenziell unendlich langen Datenströmen sowie die Notwendigkeit der schnellen und frühzeitigen Ausgabe von Verarbeitungsergebnissen.

Letzterem widmet sich die vorliegende Arbeit. Ziel ist es dabei, eine vom Anwender festgelegte Dienstgüte ('Quality-of-Service', QoS) für den Verarbeitungsprozess einzuhalten. Dabei werden zunächst geeignete QoS-Merkmale wie maximal zulässige *Ausgabeverzögerung* und minimal notwendige *Ausgabedatenrate* definiert. Danach wird ein Kostenmodell zur Berechnung der benötigten Verarbeitungsressourcen bei gegebener Dienstgüte abgeleitet. Auf dieser Basis erfolgt eine Einplanung der einzelnen Verarbeitungsoperationen; in Abhängigkeit der geforderten Dienstgüte und der im System verfügbaren Ressourcen kann das Gewicht zwischen den einzelnen Ressourcen bzw. zwischen den einzelnen QoS-Merkmalen verschoben werden.

Eine Einplanung von Ressourcen setzt genaue Kenntnis über die Charakteristik der Verarbeitungsoperationen und der eintreffenden Datenströme voraus. Da die Kenntnis der Datenströme in den meisten Fällen nur auf Erfahrungswerten beruht, ist eine Änderung der Planung und damit eine Revision der Ressourcenreservierung von Zeit zu Zeit notwendig, was zu einer Unterbrechung der kontinuierlichen Datenstromauswertung und der zugesicherten Dienstgüte führt. Das in der Arbeit vorgestellte Robustheitskonzept dient als Unterstützung für den Anwender und ermöglicht ihm, die Unterbrechungen der Datenstromauswertung durch Zugabe von Ressourcen zu verringern.

Acknowledgments

I would like to express my gratitude to the people at the database chair of our university. The excellent working atmosphere there helped me through the ups and downs during my work and made this thesis possible at all.

Most importantly, I would like to thank my supervisor, Professor Wolfgang Lehner, for nearly four years of supervision and critical as well as constructive commentary on my work. He supported me with outstanding guidance through my doctoral research, which included countless discussions on the individual aspects of data stream processing. My special thanks regarding the creation of this thesis go to Simone Linke. She was an invaluable help in revising the thesis' text modules in terms of finding typos and correcting their grammar. Furthermore, I would like to thank my colleagues Henrike Berthold, Dirk Habich and Marc Fiedler. They provided lots of motivations and suggestions during our 'coffee break discussions' and 'white-board sessions'. Also, this work would not have been possible without the help of all the students who worked at the database chair or dedicated their seminar papers or diploma theses to issues in the data stream research area. My thanks to all of them!

Finally, I thank my parents. They encouraged me to choose my own way in life, to make my own decisions and to bear the consequences. They stood by me in my personal as well as in my academic life. Therefore, I dedicate this thesis to them as a small symbol of my gratitude.

Dresden, October 2006

Sven Schmidt

Contents

1. Introduction

Processing data from a variety of different heterogeneous sources has gained more and more importance over the last years. For this reason, nearly all commercial database vendors have added support for managing distributed data originating from different sources. Nevertheless, Database Management Systems (DBMS) aim at storing or at least registering and indexing a more or less static data repository, which is said to be of *permanent* nature. If data sources continuously disseminate data, and if applications continually run so-called *continuous* or *standing queries* on that data, the principle of query processing of database systems does not directly meet the application's requirements.

Data Stream Management Systems (DSMS) focus on application scenarios to which the traditional query processing model of DBMS does not apply; the following three reasons can be given: First of all, the volume of the disseminated data may be so high that the cost for storing these data sets (even temporarily) would be out of scale. Second, the data set may be infinite and thus cannot be stored completely. Third, query processing results are supposed to be available to the application as soon as the first data item arrives. In such a scenario, data is said to be *transient*, whereas the application's queries are treated as *permanent*. More generally, while a DBMS works in a query-driven fashion, a DSMS puts the focus on data-driven query evaluation.

Stream Processing Example

This paragraph discusses the motivation behind data stream processing by reviewing two typical application cases selected from the previously mentioned area of interest.

In today's modern electrical train services, an uninterruptible, reliable and safe power supply from the overhead cable to the train via a current collector (sometimes also called pantograph) must be guaranteed (Figure 1.1). However, the big challenge when attempting to meet this goal is that one of the two partners, the train, is moving. Thus, it is the task of the current collector to stay in permanent contact with the overhead cable, which means that it must press strongly enough against the cable. Now, on the one hand, the pressure must not be too high because it would wear off the cable of the collector. On the other hand though, the pressure must not be too low, so that the collector will not lose contact, not even for a short time, because this could cause arcs, which then again would lead to an enormous wear-off. The goal here is to achieve an optimal pressure to keep the collector's and the overhead cable's wear as low as possible.

There are numerous causes which may hinder the implementation of this requirement. This includes varying distances between the train and the overhead cable, or crosswinds which move the overhead cable or set it swinging. Today's current collectors are complex

Figure 1.1.: Overhead Cable with a Current Collector

mechanical systems, which are the result of long-lasting research activities and countless experiments. In this context, one also refers to the adjustment of the current collector as *passive controlling*. However, mechanical systems have their limits, which are attributed to the laws of nature. These limitations can be illustrated by a simple example: A high-speed train runs at 250 km/h, that means it passes about 69,5 m per second, or vice versa, it takes only 14,5 ms to pass one meter. The problem is that a passively controlled current collector is too slow to react in such a short time if an adjustment becomes necessary.

One possible and promising solution might be an *actively controlled* current collector. That means that the entire train is equipped with a sensor network, which measures different environmental and technical conditions, including the distance between the train's top and the overhead cable. All those sensors produce a constant stream of data, which has to be processed and analyzed before any further calculations and/or decisions can be made. That means in detail: If it is assumed that the distance, for example, is measured four meters before the current collector passes that particular place, there will be less than 60 ms time to process and analyze the data stream produced by the sensors. In addition, the calculation for the optimum pressure must be done within this time, as well as the adjustment of the current collector itself. A DSMS might be the first choice for that purpose for several reasons, such as low running costs, or more flexibility compared to a hardware-based solution. Moreover, a software solution is easier to administrate and to adapt to changes ([SLL05]).

As another example, consider an industrial process of casting metal workpieces. To achieve optimal results in the casting quality and in the lifetime of the casting molds, modern foundries currently experiment with equipping the casting mold with sensors. Figure 1.2 illustrates the experimental arrangement of such an industrial processes. The casting mold on the left side is equipped with sensors. The sensors mainly measure temperature and pressure (analog signals) at different points of the casting mold and many switcher signals (digital signals) describing the progress of the casting procress ([SFL05]). These signals form a data stream which is to be queried by a DSMS for filtering the relevant data. The DSMS can be run by an ordinary computer system, which is shown on the right side of the figure.

To keep up with the amount of measured sensor values, the DSMS is supposed to provide a minimum data rate at which it can process the analog and digital signals.

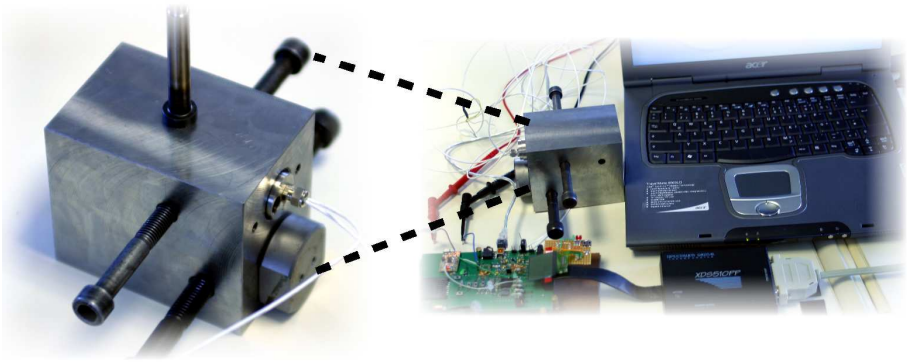


Figure 1.2.: Casting Mold Sensor Equipment

Otherwise, if sensor values get lost in situations of high load, the costly experiments would have to be repeated, as the observed data sets are incomplete.

Based on the motivation of these and many other examples, systems suited for processing transient data streams have been developed during the last years. These Data Stream Management Systems are based on an architecture slightly different from a DBMS, which allows to satisfy the application's needs mentioned above. Therefore, existing DBMS concepts are exploited on a conceptual as well as on an implementational level; this includes data models as well as the concept of logical and physical operators or query optimization strategies.

To put it into a global perspective, Data Stream Management Systems can be seen as a supplement to Database Management Systems (Figure 1.3). Within this global picture, the DSMS takes on the role of a *pre-processor* for incoming data. If an answer to a continuous query is required as a reaction to the incoming data, the DSMS may return the information to the application in an immediate and continuous fashion. In addition, *lookup queries* to the central repository may be necessary to weave permanent data into query processing. If the DSMS only plays the role of a filter regarding the DBMS, the pre-processed data (often reduced in cardinality) is finally stored in the DBMS repository for being queried by the user at a later time in an *ad-hoc* manner. Using this concept, the data management facilities are tailored to the specific application scenario's requirements.

Up to now, a variety of research activities have been directed towards different aspects of managing data streams. Data Stream Management Systems like Aurora [TcZ+03], STREAM [BDM04], PIPES [KS04], Gigascope [CJSS03b], TelegraphCQ [CCD+03] and NiagaraCQ [CDTW00] have been developed within the last decade. Within the context of this thesis, the QStream DSMS ([BSLH05, SFL05]) acts as implementational framework and environment for conducting experiments.

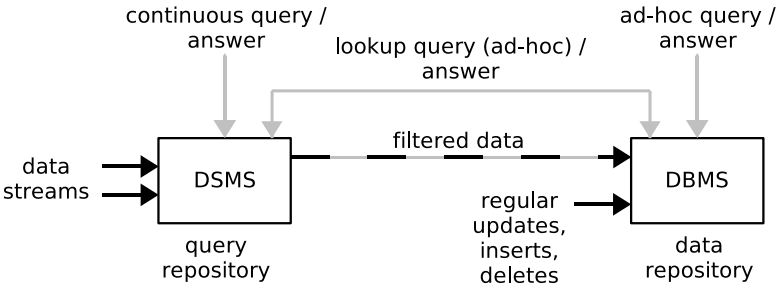


Figure 1.3.: DSMS versus DBMS

DSMS Challenges and Motivation of This Thesis

The challenges imposed by Data Stream Processing Systems are manifold. Survey papers like [GÖ03b], [BBD+02] and [KS03] present a good and broad overview on this topic. Two crucial points need to be mentioned. First, DSMS operators must not be blocking, and second–even though the incoming data streams are infinite–operators are only allowed to maintain an internal state of limited size.

The former requirement generally means that, if data continuously flow into the data stream system, the user also expects query results in a continuous fashion. Operators like join and aggregation, which are based on blocking implementations in database systems, must be available in an unblocking version. That is, a join must produce output tuples even if it has not yet seen the entire input. Also, the aggregation operator has to output the aggregate value even if the data source still delivers data.

The reason for the requirement of limiting the size of the internal state is the finite storage capacity within a DSMS. Therefore, historical data is often stored at a coarser granularity, which enables a trade-off between storage capacity and the accuracy of the query result.

In addition to the general DSMS requirements, Quality-of-Service management (in conjunction with real-time data stream processing) is of particular interest and thus receives the focus of this thesis. Within database systems, data is acquired by the DBMS depending on the speed of the query processor (*pull-based* processing). In comparison, data stream processing works in a *push-based* manner. On that basis, real-time processing–in a broader sense–means that the DSMS has to adapt to the data delivery characteristics of the sources to come up with the arriving data ([ScZ05]). In a narrower sense, this means that data stream processing may be used in time-critical environments like production control or sensor data acquisition. There, stream processing must be fast and predictable regarding the timeframe. For that reason, the user is supposed to specify Quality-of-Service (QoS) requirements along with the respective standing query. These requirements instruct the DSMS how fast the standing query is to be evaluated or what fraction of the input data is of particular interest and should therefore not be dropped during overload situations.

The use case of the train's current collector and the industrial data acquisition process are such applications with typical QoS requirements: Working with hard time limits should guarantee that the standing queries can be executed in time. The sensor data are to be processed with a *minimum data rate*, which is oriented at the train's speed and the casting mold sensor data rate, respectively. Furthermore–within the train example–the adjustment of the current collector's pressure against the overhead collector has to take place very shortly before the collector passes the particular place. Thus, a *maximum output delay* is also given as a QoS requirement for the standing query, which calculates the current collector's pressure from the measured sensor values.

Structure and Organization of this Thesis

The fundamental guideline for the main part of this thesis is based on the *operational perspective* of Data Stream Management Systems (Figure 1.4). It incorporates an explicit distinction between different optimization steps and strategies with regard to their purpose and the time at which they are applied. Individual DSMS optimization approaches are compared while considering their impact on the resource consumption and Quality-of-Service management. The latter receives particular attention because–due to the data-driven query evaluation–it imposes new challenges compared to DBMS query processing. Now, each operational step is described in conjunction with its impact on resource consumption.

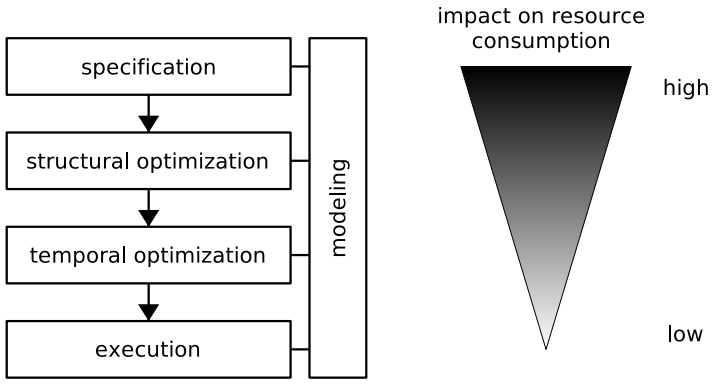


Figure 1.4.: Overview of the DSMS Operational Perspective

- **Modeling:** The evaluation of standing queries has to follow a certain data processing model. This includes the classification of Quality-of-Service metrics as well as the definition of data streams, standing queries and basic operators. The subsequent optimization steps and the execution phase build on these definitions for creating effective strategies and managing runtime activities, respectively.
- **Specification:** Prior to the query evaluation, the data streams have to be registered at the DSMS along with their type or their schema. In addition to that,

standing queries have to be specified by the user and added to the query pool of the DSMS. Existing systems offer different possibilities, for example, SQL-like declarative query languages or procedural methods like the specification of the dataflow using a graphical query representation. Additionally, some QoS requirements or preferences may be given along with the standing query.

During the step of the input specification, the DSMS workload, and thus the resource consumption, is influenced the most. The number of concurrently registered queries, the query complexity, and the data streams with their typical properties (like arrival rate and bursts) form the basis for any resource estimation or calculation approaches.

- **Structural Optimization:** During the structural optimization, the specified queries are transferred to some kind of internal representation (query graphs, for example). Different optimization techniques are applied on that basis. Eventually, they lead to a structurally optimized query network (containing all the standing queries which are to be evaluated). Optimization goals in this context are manifold. For example, the focus may be on increasing the performance or on lowering the resource consumption.

- **Temporal Optimization:** Temporal optimization is exclusively applied in data stream systems. It takes a query network whose structure has previously been optimized and decides for each component *when*, *how long* and *how often* it should be executed to continuously produce results for the given standing queries. The temporal optimization may be directed towards different goals, which should conform to the optimization goal of the former structural optimization.

- **Standing Query Execution:** Standing queries are evaluated continuously after they have been specified and optimized. Aside from the query evaluation, the DSMS has to collect statistics of the data stream and query processor in order to adapt to new environmental situations if the statistics change too much. In general, the better the DSMS considers and exploits the statistics during the execution, the lower the resource consumption will be. Contrary, the overhead of runtime activities itself consumes resources and is therefore subject to regular optimization approaches.

The structure of this thesis is illustrated in Figure 1.5. After this introduction, Part I gives a comprehensive overview of the state of the art in data stream management. It starts with fundamental DSMS modeling aspects and general QoS management issues. Thereafter, the related work part is divided into sections on standing query optimization techniques (Chapters 3 and 4). The resource management techniques leading to the appropriate optimization goal are interleaved appropriately. The related work part ends with a review of re-optimization and adaptation techniques which are to be applied at runtime during query evaluation. Last but not least, approximated query results for lowering the resource consumption are discussed. Their connection to the work of this

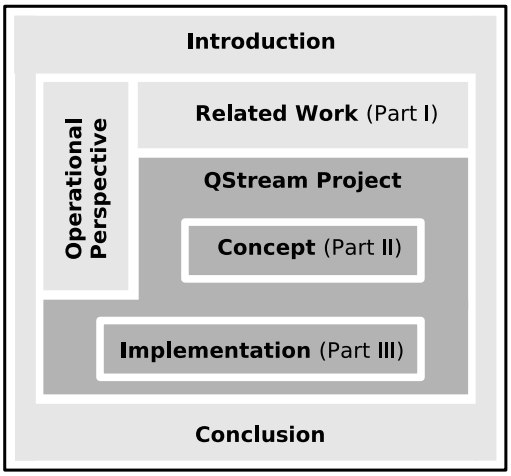


Figure 1.5.: Structure of This Thesis

thesis is stressed during the related work chapters by making a forward reference to the respective chapter of the thesis' main part.

Part II contains QStream's novel concepts for realizing QoS guarantees within a DSMS. It follows the operational perspective with the exception that the structural optimization aspect is not covered by QStream. Part II starts with presenting the *date stream model*, the *QoS model* and the *query and operator model*, respectively, in Chapter 6. Thereafter, Chapter 7 announces the individual *QStream operators*. The a-priori *resource calculation approach* as well as the *scheduling strategies* allowing for query evaluation with QoS guarantees are combined and presented in Chapter 8. At the end of Part II (Chapter 9), the runtime management of QStream is discussed in the context of the *robustness concept*.

The implementation part (Part III) contains, first, the description of the QStream prototype along with its program components and the sensor data acquisition concept (Chapter 10). Then, Chapter 11 provides the experimental evaluation of the cost model, scheduling strategies and the robustness concept.

The thesis concludes with Part IV. It summarizes the main contribution and stresses how the challenges of QoS-guaranteeing data stream processing are fulfilled with the proposed concepts in terms of modeling, resource management, scheduling and robustness.

Part I.

Related Models and Systems

2. Fundamental Prerequisites for Data Stream Processing

This section covers the modeling aspects of data stream management systems in an informal way. The goal is to give an overview of the basic structures of the streaming data as well as of query specification techniques including the concept of annotating QoS requirements at standing queries. First, Section 2.1 describes data streams from different points of view. Second, Section 2.2 discusses the concept of the standing queries. Third, Section 2.3 provides an overview of Quality-of-Service management and an appropriate classification of data stream systems. Finally, the optimization approaches of the following chapters are announced in Section 2.4.

2.1. Data Stream Modeling Aspects

Intuitively speaking, a stream is a potentially infinite sequence of data items. Each of the data items is annotated with timestamp information from a completely ordered and discrete time domain ([BDE+97, MWA+03]). The order of the stream elements is defined by the order of the timestamp attribute. A data item may be timestamped either by the data source when it is produced or by the DSMS when it arrives. In the former case, data items may arrive in disorder due to transmission latencies. Furthermore, in both cases, there may be duplicates with regard to the timestamps, depending on the chosen time granularity.

The following subsections go into more details by discussing, first, the layout of single data items, second, the temporal relationship of consecutive data items, and third, the evaluation of the data items' attribute values over time.

2.1.1. Internal Structure of Data Stream Items

The structure of data stream items is classified in two independent ways.

First of all, items change their format and content on the way from the data source to the data sink (transformation). For example, a stream containing items directly from the data source is denoted as *raw* in PIPES ([KS05b]) and as *base stream* in STREAM ([ABW03]). As soon as some kind of pre-processing is performed, the stream changes to a *physical stream* or a *derived stream*, respectively. A similar concept is used in Gigascope ([CJSS03b]). There, a two-level query architecture is established, where *low-level queries* work on raw packet sources and hand over pre-processed data streams either to an application or to *high-level queries*. The benefit of this data stream or item classification is that operations dedicated to a specific kind of input data can be defined.

Furthermore, the low-level queries of Gigascope can exploit the network card hardware capabilities.

Second, different views on the data stream items may be established, which allows for abstraction from implementational details of the data exchange. This is similar to the concept of defining different protocol layers as views on network traffic. Within the related work, a *physical* and a *logical* view on data items is distinguished.

Within the *physical view*, a data item consists of a timestamp and an arbitrary portion of additional data. The physical view allows the support of *delta data items*, that is to say, a data item may provide full information on a sensor value or any other real-world event, or it may only provide the difference to the item that arrived last. As an advantage of the latter, the transmission requires lower bandwidth. A disadvantage lies in the increased processing complexity because each operator has to reconstruct the complete data item value or–alternatively–the reconstruction of an item is done only once per item at the beginning of the DSMS processing.

In current DSMS, the approach of incremental transmission of data is used in the field of XML stream data processing (SPEX [OFB04, OMFB02]) where the items are complex XML data structures and updates are considered for processing, too. Similarly, the approach of the Borealis DSMS for invalidating tuples retrospectively ([AAB+05]) incorporates an incremental update mechanism with the help of deletion or replacement messages. Both of these approaches aim at breaking with the append-only nature of data streams. Furthermore, the SPEX approach saves bandwidth by avoiding the repeated sending of the whole XML document. The larger the data items, the higher the bandwidth savings. For example, in the field of multimedia data streams, bandwidth can be reduced dramatically by transmitting only changes of objects (like MPEG layers, described in [NS95]).

Aside from the physical view, the *logical view* facilitates the correct interpretation of the data item, which has to be based on a certain data model and has to follow a certain schema. For example, many DSMS (Aurora, STREAM, Gigascope etc.) consider data as relational tuples, whereas other DSMS (TelegraphCQ, StreamGlobe, NiagaraCQ etc.) focus on processing XML data. In the former case, the tuples must be based on a relational schema, whereas in the latter case, XML fragments should be valid regarding a DTD or an XML schema. Without loss of generality, one uses the term 'tuple' as a synonym for the items of the logical layer. A definition for logical-layer data is given in [ABW03], where a tuple is denoted as $< s, \tau >$, with s being the tuple's content following a certain (relational) schema and τ being the associated timestamp.

2.1.2. Temporal Relationship of Consecutive Data Items

Stream items may be closely related to each other with regard to a *(temporal) session*, which is implicitly marked within the data stream. A session is characterized by certain elements signaling the beginning and the end of a real-world event. The advantage of sending the tuples individually instead of waiting for the event's completion (and building one larger data item/object) is the opportunity for producing early results, even if the event (such as a telephone call) is a long-lasting one. Two examples for the

incorporation of this concept based on the example of AT&T call records are Hancock [CFPR00] and NESTREAM [CA04]. Both provide a session layer by connecting the individual tuples from the logical stream layer. While Hancock deals with relational-like data, NESTREAM focuses on XML fragments.

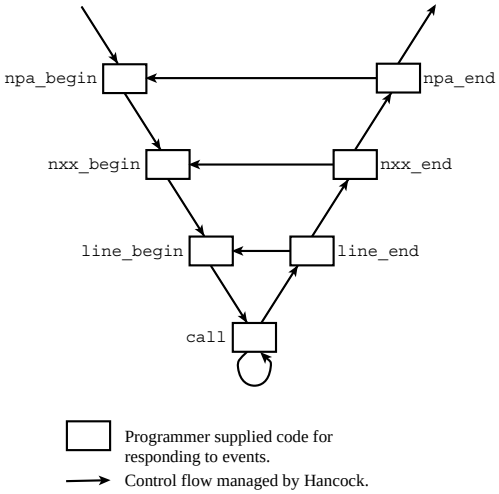


Figure 2.1.: Session Constitution in Hancock ([CFPR00])

Figure 2.1 illustrates the events belonging to the recorded AT&T telephone call data. The events *npa_begin*, *nxx_begin*, *line_begin*, *call*, *npa_begin*, *nxx_begin* and *line_begin* belong to a telephone call and are defined by Hancock's event-detection functions. First, a new area is detected (*npa_begin*), followed by a new exchange (*nxx_begin*) and a new phone number (*line_begin*). The bottom-level event is the individual *call*. If the last call record of a phone number has been seen, the *line_end* event is triggered. For reacting to the various data stream events, actions for responding to the events are specified.

Similarly to a session, the beginning of new episodes of data delivery can explicitly be marked by the insertion of *punctuation messages* into the data stream. This concept was introduced by [TMSF03]. A sample scenario is an online auction, where punctuation messages indicate that no more tuples with bids appear in the data stream, when an auction item has finally been sold.

More formally, punctuations are annotations embedded within a data stream, which annotate the end of a specific subset of data. They can be seen as predicates on the stream elements, which have the form of an ordered set of patterns, with each pattern corresponding to an attribute of a tuple. The punctuation predicate must evaluate to false for every stream element following the punctation. Thus, the former infinite data stream can be viewed as a mixture of concatenated finite streams.

The punctuation concept was implemented in the NiagaraCQ DSMS, where punctuations can be defined for XML data. The operators union, group-by and join have been extended to consider the punctuation semantics.

To compare the two concepts, sessions aim at representing complex (nested) real-world events, whereas the main intention of punctuation messages is to support stream operations to work on finite data sets. For example, with the help of punctuation messages, it is now possible to unblock operators like join or aggregation. To emphasize the latter, [GÖ03b] punctuations are classified as constraints on the data level because they appear as physical items in the data stream. In contrast, so-called schema-level constraints (like k-constraints of STREAM) are discussed later in Section 2.2.4 in conjunction with allowing a bounded disorder of the stream elements.

Another concept for annotating the temporal relationship of tuples, *heartbeat* messages ([SW04a]) can be inserted into the data stream. The heartbeats are sent by the data sources even if no or only a few data items are produced. That is, heartbeats can be used for providing fault tolerance. They signal that data sources within a distributed DSMS are still alive, even if they do not deliver any real data at the moment ([ZSC+03]). Furthermore, the heartbeat information can be used to ensure that–if out-of-order arrival is assumed–all tuples are sorted by their timestamps before they are sent to the query processor ([SW04a]): Heartbeat messages from the data source including a timestamp τ indicate that no more tuples with a timestamp smaller than τ will arrive from that source. That is, the input tuples are buffered and sorted (following the given contraints of out-of-order arrival) and then handed over to the query processor, even if no actual input data has been delivered. The incorporation of the heartbeat mechanism into the Gigascope ([JMSS05]) DSMS aims at unblocking multi-stream operators such as join and merge. The heartbeat messages–which are generated by low-level query nodes and propagated upwards the query graph–signal each high-level operator the progress of the stream, even though no tuples might arrive at the specific operator (e.g. due to previous filtering operations). Based on this additional information, multi-stream operators can continue processing even if no input data arrive at one of their inputs.

2.1.3. Attribute Evaluation Over Time

Aside from the temporal relationship of consecutive data items, a classification of the attribute evaluation over time would be helpful to provide proper application semantics. For that reason, this thesis provides a classification of so-called partial data streams, which will be discussed in the model chapter of QStream (Chapter 6) in Part II.

2.2. Characteristics of Standing Queries

Now, the operations of a DSMS are reviewed on the basis of the data stream model. It ranges from the query specification to individual DSMS operators. This section opens with a description of the specification of standing queries, which is based on the illustration in Figure 2.2. Then, the extensions of standing queries to account for transient data streams are summarized in Section 2.2.2. Finally, one 'drills down' and the specifics of DSMS query plans (Section 2.2.3) as well as the elementary operators of the various data stream systems (Section 2.2.4) will be discussed. The points mentioned above, together

with a discussion of existing QoS metrics in Section 2.3, are prerequisites for realizing non-blocking, QoS-capable DSMS operators.

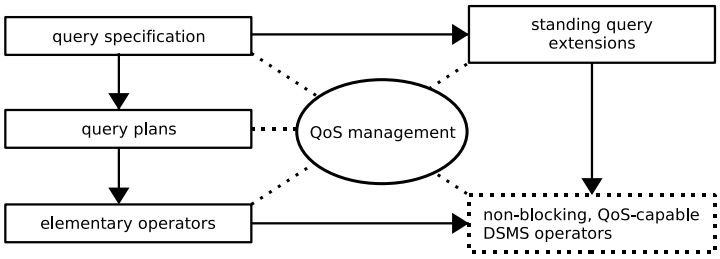


Figure 2.2.: Query Extensions for DSMS

2.2.1. Type of Query Specification

Most DSMS rely on the user's knowledge and experience with query languages of database systems and build on their specific query approaches. A first classification for standing queries is given in [GÖ03b]; they distinguish between:

- **Relation-based languages:** Data streams are treated as infinite relations (with a timestamp attribute). Example languages are CQL in STREAM [ABW03], StreaQuel in TelegraphCQ [CCD+03] and AQuery [LS03].
- **Object-based languages:** Here, data sources are treated as hierarchical (Tribeca, [Sul96, SH98]) or abstract data types (COUGAR, [BGS01]) containing a timestamp attribute.
- **Procedural languages:** In comparison to the former declarative query language approaches, a procedural query specification directly orchestrates the data flow. An example is the Boxes-and-Arrows principle from Aurora [CcC+02].

2.2.2. Standing Query Extensions

The evaluation of standing queries requires additional information or instructions for the query processor. It is independent from the query specification of the previous section. The *extensions of standing queries* are described in the following.

- **Selecting the relevant stream fragment:** The challenge of processing an infinite data stream is met by separating the stream into portions which can be processed one at a time. That is, a *window* is defined on the stream data.

  A first window classification is given in [GÖ03b], where the window's beginning and its end are defined either on the basis of the number of tuples (*logical window*) or directly on the basis of the tuples' timestamp information (*physical window*).

  Further classification criteria are the movement of the window's endpoints and the window update interval. Using a *fixed window*, both the start and the end

timestamp do not move. In contrast, a *sliding window* conceptually moves over the data stream, and a *landmark window* is constrained by one fixed and one moving endpoint. Depending on the update interval, the window either conceptually slides over the data stream tuple by tuple (*sliding window*) or it moves by a larger number of tuples (*jumping window*).

With this concept, stream fragments ranging from a single tuple to a large sequence of tuples may be specified. The length of a window depends on the specific application requirements. The amount of tuples kept in the window influences the required memory resources for the query evaluation considerably. The window's length should be determined by either of the following two factors:

- Calendar-oriented window length: In this case, the window length is defined by the user who is aware of the semantics of the query result with the chosen window length. The basis are calendar-like time spans such as month, day or even hour, minute and second; a possible scenario is the evaluation of business data streams.
- Quality-driven window length: If sophisticated stream processing algorithms are applied (e.g. to perform signal processing on continuous data streams), it becomes difficult for the user to specify the window's length because the impact on the result is not as clear as in the previous case. In such a scenario, abstract measures, e.g. the operation-specific result quality, determine the window length. For example, the length of an aggregation window of sensor values is influenced by the allowed deviation of the aggregate value. In [BSLH05] and [Haa97, HHW97], it is shown that for given characteristics of the sensor values (minimum and maximum) and an upper bound for the deviation of the aggregate value (which is held with a certain probability), the required (worst-case) length of the aggregation window can be calculated. In general, the quality of the operation's result improves with an increasing number of tuples within a window. Also, interpolation and resampling operators benefit from larger window sizes: If more tuples are available for computation, more complex interpolation functions can be applied to produce more accurate results.

  Window size as a Quality-of-Service criterion is described later in Section 2.3. Moreover, it involves aspects of approximation and is therefore topic of Section 5.3.

- **Specifying periods of query evaluation:** Following [GÖ03b], a standing query may be evaluated either in a data-driven way (*streaming*) or *periodically* with a user-determined execution frequency. If a query is to be evaluated only once (at a specific point in time), it is denoted as *ad-hoc* query.

- **Choosing initial sampling rates:** Some DSMS allow the specification of an initial stream sampling rate to cope with the stream load in a push-based processing

model. This comes along with a reduction of the result quality and is discussed in the next paragraph.

- **Annotation of QoS requirements:** If the DSMS supports Quality-of-Service parameters for query evaluation (beyond the window length), the DSMS should enable users to specify their quality requirements along with the standing queries. The QoS specification may be annotated like the sampling clause in STREAM (`Sample(2)`), which describes that, on average, every n-th (in this case every other) tuple is passed through the sample operator ([MWA+03]). As another example, sliding window size requirements in PIPES are specified as a range within each query (`[Range Min 2 hours, Max 5 hours]`, [CKSV06]). In more abstract terms, Aurora requires a graphical specification in form of different QoS graphs (Figure 2.3) for each result of standing queries ([CcC+02]). The delay-based QoS graphs describe how the QoS of a query answer slowly degrades with increasing delays of the result. Similarly, the drop-based graph indicates how the result quality depends on the amount of dropped tuples. Finally, with the value-based QoS graph, the user signals which value range is of most importance and will thus lead to the highest QoS. For example, the temperature values within a critically high range are more important for fire detection than room temperature values.

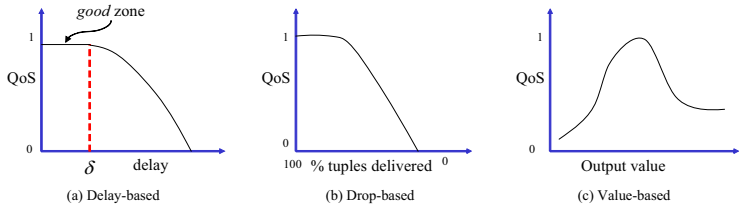


Figure 2.3.: Specification of QoS Diagrams in Aurora ([CcC+02])

2.2.3. Query Plans

Once a standing query has been specified and submitted to the DSMS, it has to be transformed into a *query execution plan (QEP)*. This is done in a way that is analog to the procedure for database systems. First, a logical query plan is created based on operator algebra, and second, a physical query plan or dataflow plan is constructed by choosing and inserting physical data stream operators. The benefit of this stepwise query translation is the abstraction from physical details on the one hand, and the potential optimization on the logical as well as on the physical layer on the other hand. The general procedure is closely related to query translation and optimization in regular database systems ([Gra93]). In the following, specific examples for query representations on the logical and on the physical layer are given.

- **Query representation on the logical layer:** On the logical layer, queries are represented as expressions using appropriate algebra of the particular DSMS. The

basis is a set of logical operators. The specified query determines which operators are used and what the algebraic expression looks like. For example, Aurora ([ACc+03]) uses Stream Query Algebra (SQuAl), which consists of seven primitive operations. These are *Filter*, *Map*, *Union*, *Sort*, *Aggregate*, *Join* and *Resample*. The logical query specification is done on a graphical basis where the operator boxes are connected using edges. As another example, STREAM uses a 'logical plan generator' to transform Continuous Query Language (CQL) expressions into expressions of relational algebra. Some CQL-specific operators are included for expressing windows and transforming streams into relations and vice versa ([MWA+03, Sta04]).

Finally, the PIPES DSMS uses temporal algebra ([KS05b]) for the query representation. The logical operators are defined with logical operator algebra. A dedicated sliding window operator ω_w^s restricts the validity (time interval) of each tuple of the stream to the value w. A logical query expression to perform a selection on two windowed streams followed by a union operation may be given as

$$\sigma(\omega_w^s(S_1^l)) \cup_+ \sigma(\omega_w^s(S_2^l))$$

where, first, a window is applied to both input data streams S_1^l and S_2^l; second, an independent selection σ is performed on both window streams; and finally, the (schema-compliant) partial results are merged using the union operator $\cup_+$.

[GV04] use logical level expressions similar to the extended relational algebra and provide seven logical operators. Each of the logical expressions is implemented as a set of dataflow expressions (on the physical layer). Windows are not considered in expressions on the logical level. In the following example, the input streams r with schema (ab), s with schema (bc) and t with schema (ac) are joined

$$r(ab) \bowtie_b s(bc) \bowtie_{ac} t(ac)$$

- **Query representation on the physical layer:** If the logical query plan has been selected (and possibly optimized), it then is transformed into a physical one. In STREAM, for example, this is done by the 'physical plan generator.' A physical query plan is a network describing the data flow from the data sources to the data sink. It contains physical operators, FIFO queues for connecting them and occasional repositories for the materialization of intermediate results. Now, the physical query plans are discussed for the DSMS examples already mentioned in the previous paragraphs.

  First, the logical level expression example $r(ab) \bowtie_b s(bc) \bowtie_c t(ac)$ from [GV04] is translated into a set of dataflow expressions. Three alternative paths leading to the same query result are given. The consecutive dataflow expressions of each single query path are listed, separated by commas:

$$\delta_r :\rightarrow (w_r), \bowtie_b (w_s), \bowtie_{ac} (w_t), \rightarrow (w_{out})$$

$$\delta_s :\rightarrow (w_s), \bowtie_b (w_r), \bowtie_{ac} (w_t), \rightarrow (w_{out})$$
$$\delta_t :\rightarrow (w_t), \bowtie_c (w_s), \bowtie_a (w_r), \rightarrow (w_{out})$$

In this specific case, the expression $\rightarrow (w_r)$ denotes that–during a 'recorder operation'– the result of the previous step is written to a window w_r, which has the meaning of a synopsis. The operation $\bowtie_b (w_s)$ represents a join operation of the previous result with the specified window (w_s). The symbols δ_r, δ_s and δ_t specify the three input data streams, and the following join and window expressions describe the data flow strategy. Finally, the result is written to the window w_{out}. Due to the join associativity, three different execution plans exist. One option would be to write the input of streams δ_s and δ_t to the windows w_s and w_t, respectively, and to evaluate them in the order of the set of dataflow expressions shown in the first line:

$$\delta_r :\rightarrow (w_r), \bowtie_b (w_s), \bowtie_{ac} (w_t), \rightarrow (w_{out})$$
$$\delta_s :\rightarrow (w_s)$$
$$\delta_t :\rightarrow (w_t)$$

The resulting and semantically equivalent data flow graph, including all execution alternatives, is illustrated in Figure 2.4.

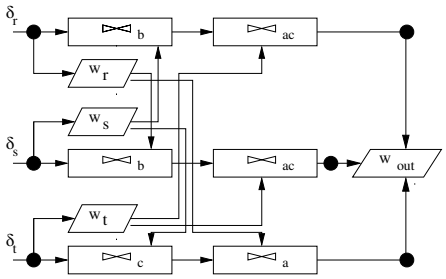


Figure 2.4.: Example Dataflow Implementation ([GV04])

Within the STREAM DSMS, a physical query plan consists of operators, queues and synopses (Figure 2.5). For example, a join operator maintains two synopses S_4 and S_5, one for each input stream (provided by the queues q_3 and q_4). Furthermore, synopses may be used to apply compression and approximation techniques to reduce the resource consumption (Section 5, runtime management) as well as to enable multiple standing queries to share intermediate results (Section 3, structural optimization) with the help of so-called 'stubs.' In the example, the synopses S_1, S_3, S_4 and S_2, S_5, respectively, share their contents.

As a final example, Figure 2.6 shows the physical query plan components of the Aurora system. The three different paths belong to different types of queries: the topmost path represents a standing query, the middle path shows a view, and the

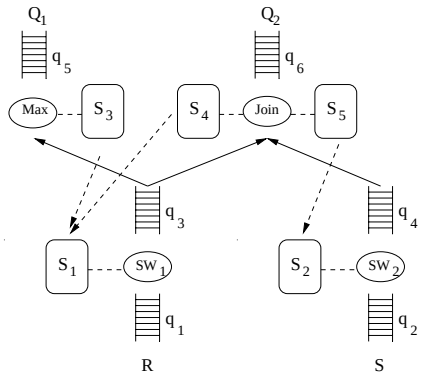


Figure 2.5.: Query Plan Components of STREAM ([ABB+03])

bottom path stands for an ad-hoc query. All query types are based on the same query plan components. They include operators, connection points (dark circles; they allow for the connection of boxes and offer storage capacity for data of the recent past), storage capacities independent from connection points (like $S3$ in the view path; this can be seen as a materialized view), and QoS specifications at the end of each path.

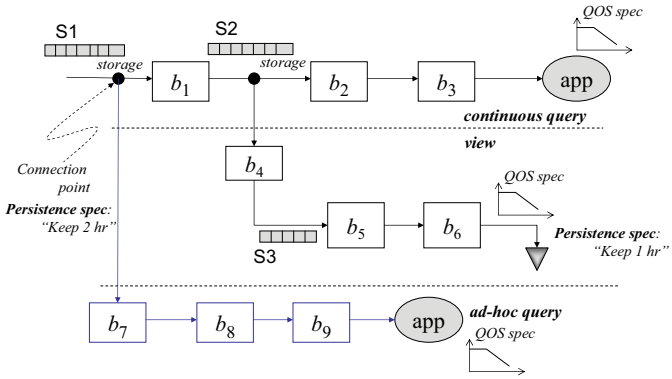


Figure 2.6.: Query Plan Components of Aurora ([CcC+02])

To summarize, data stream systems allow to distinguish between different layers of the query representation with the goal of optimization and abstraction. In comparison to DBMS, components like synopses, buffers and connection points are explicitly required to support the transient nature of data streams as well as of multiple standing queries which are to be evaluated concurrently.

2.2.4. Operators

Query plans are composed of elementary operators. They provide basic functionality with the goal of maximum reusability. In the next sections, the operators are first classified depending on the necessity of maintaining an internal state. Then, their order sensitivity is discussed, and finally, some example operators are provided which are novel in the context of Data Stream Management Systems.

- **Operator state characteristics:** A first classification criterion of DSMS operators is the necessity of maintaining an internal state for successful operation. A *stateless operator* is able to immediately produce a query result without accessing any kind of internal structures and without waiting for the arrival of future tuples. Examples for stateless operators are selection, projection, and map.

  In contrast, a *stateful operator* has to maintain a state using a storage structure (synopses). For the storage structure, efficient access must be guaranteed (for example, with the help of indexes). An operator may have access to one or more synopses. For example, a binary window join operator has to maintain the state of two windows on the input streams, whereas the aggregation operator only needs a single synopsis for buffering input tuples.

- **Order sensitivity:** If a specific operation depends on ordered input data, operators are classified to be either *order-agnostic operators* or *order-sensitive operators* ([ACc+03]). An order-agnostic operator is able to process tuples in the order in which they arrive, which is true for all stateless operators, as they process the tuples independent from each other. Order-sensitive operators require ordered input for an execution to produce valuable results. Examples are stateful operators like sort, aggregation, and join. Also, a bounded disorder may be tolerated for order-sensitive operators; this has to be specified as a *schema-level constraint* and indicates an upper bound for the slack in terms of timestamp differences in the tuple order (scrambling bounds [GÖ03b], ordering constraints [ACc+03], and k-constraints [BSW04]).

- **Relational Operator adjustment:** To account for the specific requirements of data stream processing (non-blocking behavior and limited internal state), relational operators have been adjusted. Particularly, join and aggregation have been adapted to cope with the unbounded nature of the data streams.

  Join operators are extended to maintain windows of input tuples, to regularly update these windows following a specific update strategy (for example FIFO or age curves, [SW04b]), to check new tuples against existing content of the partner window (either symmetrically or asymmetrically, [KNV03]), and to use new similarity metrics to find matching partner tuples ([KS05a]). Generally, a stream join based on windows of the input data can only produce an approximate result ([DGR03]).

  Similar to joins, it is a challenge for aggregation operators to handle the continuous and possibly infinite data streams ([SW04b]). A window of input tuples contributing to an aggregate value must be maintained because the input data is only seen

once. If there exist annotations within the data stream which signal the end of the aggregation group (sessions, punctuations, heartbeats), approximate results are produced only. For aggregation operations involving large time spans, the granularity of the data may be reduced to save some storage space within the synopses ([ZGTS03]); this constitutes a trade-off between storage space and aggregation detail.

Aside from the extension and adaptation of traditional operators, various new operators have been introduced. For example, the data rate reduction is an important issue for nearly every DSMS implementation, and thus, sampling and load-shedding operators have been proposed and implemented ([TcZ+03, BDM03, BDM04]). In order to combine the querying of streams and of relations at the same time and to re-use existing relational operators, [ABW03] the mapping is described with stream-to-relation and relation-to-stream operators.

To account for the temporal nature of the stream items in particular, operators like 'coalesce' and 'split' are used by PIPES ([KS05b]). Coalesce merges value-equivalent stream tuples with adjacent time intervals into a single tuple with a larger validity interval. This decreases the data rate but enlarges the validity of the corresponding tuple, which may lead to higher memory consumption of stateful operators. In contrast, the split operator splits up a stream element into several value-equivalent elements with adjacent time intervals at the cost of an increased data rate. Thus, coalesce and split may be used during the optimization process.

To bridge the gap between processing streams consisting of independent tuples and more or less continuous signals, Aurora ([ACc+03]) proposes a 'resample' operator, which is similar to the resample helper operator which will be presented in this thesis in Chapter 7. With that concept, tuple values which existed in the real-world scenario between two tuples can be reconstructed approximately using an interpolation function. Furthermore, the QStream tuple interpolation concept forms the basis for join processing on continuous data streams. Thereby, the attribute values of the join input streams can be reconstructed for an arbitrary point in time, which is an attempt to achieve maximum result consistency.

2.3. Quality-of-Service in Data Stream Processing

If quality-aware data stream processing is required, different aspects ranging from the QoS specification (and annotation) to the implementation of the operator execution must be considered. Within this section, first, the meaning of 'better' results is discussed while introducing different QoS metrics (Section 2.3.1). Second, one differentiates between two general ways of QoS support and classifies existing DSMS on that basis (Section 2.3.2).

Some DSMS like Aurora, STREAM, PIPES and QStream already provide useful solutions, which act as a basis for this section. Generally, the more resources are available for data stream processing, the 'better' the answers to the standing queries will be.

2.3.1. QoS Metrics Classification

Due to the widespread interpretation of the term 'quality,' a variety of QoS metrics have been introduced by existing DSMS which incorporate the notion of Quality-of-Service. Some of the QoS metrics are dedicated to specific operators; others are more generic and thus applicable to the whole operator network. As illustrated in Figure 2.7, a distinction of QoS metrics with regard to the temporal aspects of query evaluation is proposed. This is justified by the data-driven query processing: there are time constraints which have to be met to keep pace with the arriving data.

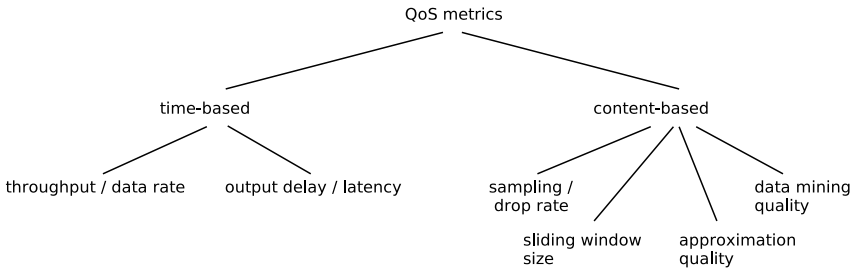


Figure 2.7.: QoS Metrics Classification

Time-Based QoS Metrics

Time-based QoS metrics receive the focus of this thesis and indicate the DSMS's ability to adapt to the push-based data delivery and to the processing speed of the DSMS. There exists a distinction between:

- **Throughput:** Throughput (or data rate) indicates the amount of stream input which a DSMS is able to handle with bounded resources. The larger the value, the higher the QoS. The user may specify a throughput requirement on a per-query basis (explicit), or the DSMS just tries to keep pace with arriving data (implicit). A variety of DSMS include throughput in their optimization strategies and therefore try to maximize the throughput with a given standing query configuration.

- **Delay:** Delay (latency) describes the time distance from the arrival of an input tuple until this very input tuple has been processed by all the operators which are in the appropriate operator path (or which belong to a certain query). Even though not every tuple may reach the output of the query network, the time from the tuple's arrival until the time of the tuple's first influence on the query result is taken as a measure. A well-established approach is to accumulate processing and buffering times of consecutive operators and FIFO queues, respectively. A smaller delay value indicates better QoS.

Content-Based QoS Metrics

Aside from throughput and delay, content-based QoS metrics exist. They do not involve temporal query evaluation aspects but focus on the problems caused by the handling of infinite streams or too high input volumes. A list of them, along with a short description, is given below:

- **Sampling rate:** The sampling rate describes the fraction of the input stream that has to be sampled out (thrown away). There is a distinction between sampling and load shedding. The former method probabilistically throws away tuples, whereas the latter is supposed to drop tuples deterministically and based on properties like the tuple's value or its priority. Load-shedding techniques are extensively discussed in [TcZ+03].
- **Output value:** Some attribute values are more important than others and thus lead to a higher result quality.
- **Sliding window size:** In Section 2.2.2, different possibilities of determining the window size of stateful operations were discussed. The larger the window, the higher the precision of the operation and the higher the resource consumption. Sliding window size as a quality metrics is explicitly introduced by [CKSV06].
- **Quality of approximation:** If stream elements have to be stored for further calculations, synopses are used to materialize the data stream content. When doing so, approximation techniques (for example randomized sketch synopses [AMS96] or histograms [JKM+98], [GG02]) are used to reduce and limit the size of the synopses. If the appropriate compression technique provides quality guarantees or probabilistic error guarantees, the quality of approximation can be considered as content-based QoS-metrics.
- **Data mining quality:** QoS measures for mining data streams are proposed by [FHKS05]. They associate different QoS metrics like *methodical quality* (which includes quality of approximation and quality of interestingness in the case of frequent itemset analyses) and *temporal quality* (which includes time ranges, time granularity and reaction time) with the stream mining techniques.

The QoS metrics mentioned above are just examples. Depending on specific operators or application areas, other non-time-based QoS metrics may be considered.

2.3.2. QoS-Oriented Classification of DSMS

There are different possibilities of negotiating and assuring Quality-of-Service requirements. This thesis concentrates on the time-based QoS metrics. Thus, Data Stream Management Systems are classified on this basis. A DSMS may either 'try its best to reach the QoS' or it may 'guarantee the QoS.' DSMS which incorporate the former property are called *best-effort systems*, whereas the latter are named *QoS guarantee systems*. The characteristics of both systems are described in the following.

Best-Effort DSMS

Based on the QoS specification, best-effort DSMS try to meet the user-given QoS requirement as closely as possible. Different runtime strategies exist which aim at increasing or maximizing the quality. With such, the preferred result quality may be specified explicitly, like it is done with the delay-based QoS graph of the Aurora system. Another possibility is to put the focus on a certain quality measure (like query result deviation) and to consider it during the optimization ([BDM04]).

Due to the system's architecture, a best-effort system is not able to give any quality guarantees on time-based QoS metrics. There may always be concurrent activities in addition to the DSMS operation which have influence on the processing times of the operators belonging to a query. Furthermore, the queries cannot be shielded effectively from each other and from concurrently running applications. Thus, other applications may be preferred (for example, due to user interaction), or too much processing time is consumed for the evaluation of DSMS queries with low importance. As a result, neither an upper bound for the delay of a single query's operator graph can be given nor a minimum or constant result data rate of a query can be assured.

QoS Guarantee DSMS

In comparison, a QoS guarantee system is able to fulfill or guarantee a *negotiated* QoS requirement during the complete lifetime of the standing query. The only prerequisite is a successful QoS negotiation process (Figure 2.8).

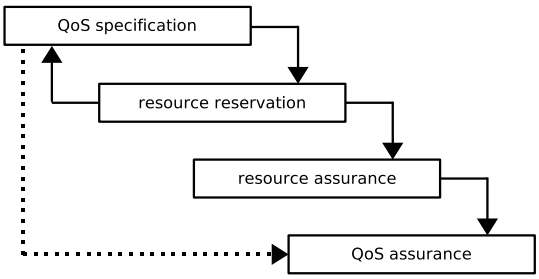


Figure 2.8.: QoS Negotiation in QoS Guarantee DSMS

It incorporates, first, the required resources for fulfilling the QoS requirements; they are calculated using an adequate cost model. In a second step, these resources have to be reserved at a resource manager and are available for use in standing query evaluations. If the statistics of the data stream do not change in an unforeseen manner, the required QoS in terms of output delay as well as the throughput can be guaranteed.

From the perspective of a system's architecture, only a real-time operating system underneath the DSMS can handle the challenge of time-based QoS guarantees. This issue is discussed in very detail in Section 4.3.3.

The two DSMS QoS approaches are compared in Table 2.1. QoS-guarantee data stream systems meet time-based quality requirements more strictly and thus clear the

way for new application areas for DSMS. Non of the existing DSMS are able to quarantee time-based QoS requirements. Therefore, this thesis puts the focus on time-based QoS guarantees and aims at establishing the *QStream* DSMS as a representative.

	Best-Effort DSMS	**QoS Guarantee DSMS**
user input	quality requirement (based on QoS metrics)	
optimization goal	maximize QoS value / approximate QoS requirement as much as possible	do not violate QoS requirement
required DSMS facilities	optimizer with appropriate strategy	resource manager, QoS negotiation, resource reservation, optimizer with appropriate strategy
required OS facilities	-	real-time support
example systems	Aurora [TcZ+03], STREAM [BDM04], PIPES [KS04], Gigascope [CJSS03b], TelegraphCQ [CCD+03]	–

Table 2.1.: Comparison of QoS Management Approaches

2.4. Perspectives of Data Stream Query Optimization

The fourth issue in reviewing the fundamental prerequisites for data stream processing is an overview of 'optimization' in the context of Data Stream Management Systems. It clearly separates *structural* from *temporal* optimizations, which are two consecutive steps regarding the operational DSMS perspective proposed in the introduction of this thesis.

First, structural optimization represents the process of arranging the operators of the operator network in the optimal way. This is mainly accomplished on the physical query representation layer; nevertheless, the logical layer acts as the basis for physical operator arrangement and should therefore be considered, too. As a result of the structural optimization, a *query execution plan (QEP)*, following a user-defined optimization goal, is created.

Second, due to the fact that within a DSMS, data must be processed in a push-based manner, the temporal aspect of the query execution is more important than for pull-based query executions in a database system. Therefore, the execution (and re-execution) order of each operator of the query execution plan should be defined more or less strictly (to continuously produce query answers, the DSMS operators have to perform their work repeatedly). This temporal optimization is based on the QEP of the previous optimization step and results in a *scheduling plan (SP)*. Both optimization issues are illustrated in Figure 2.9. Furthermore, structural as well as temporal optimizations may be repeated at runtime of the DSMS, either due to changes in the set of standing queries or due to changes in the characteristics of the incoming data stream.

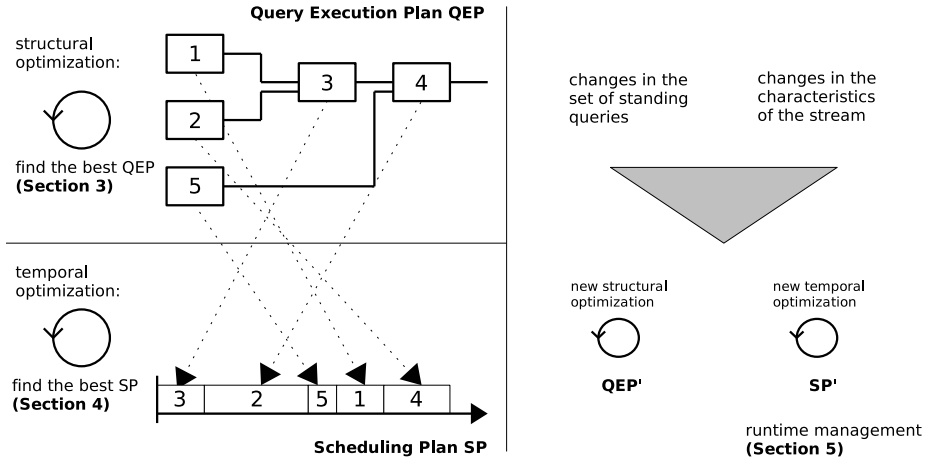


Figure 2.9.: DSMS Optimization Steps

Based on the different purposes of structural and temporal optimizations, this thesis proposes two different views of the DSMS operators: Regarding the structural optimization, the operators should be seen as *white boxes* because the optimizer should be aware of the semantics and the internally used algorithms to produce a good (optimized)

result. In comparison, operators are treated as *black boxes* during the temporal optimization phase, since only the generic operator behavior (like periodicity or runtime) and its generic input/output characteristics (like input and output rates, general selectivity, batch sizes, etc.) are of interest. In the section on related work, the terms *optimization* and *scheduling* are used when discussing optimization aspects. This is confusing in so far as often no clear distinction is made between the structural and the temporal optimization issues.

Based on [KS03] and [GÖ03a], different optimization goals can be identified, which fall into the categories *minimization of resource consumption*, *maximization of Quality-of-Service*, *efficient multi-query optimization*, and *adaptivity with regard to environmental changes*.

The first two goals are considered in Section 3 (structural optimization) as well as in Section 4 (temporal optimization). Multi-query optimization is covered in Section 3, and issues of adaptivity are discussed in Section 5 (runtime management). Time-based Quality-of-Service guarantees are not considered as optimization goal by recent related work. For that reason, they will be intensively discussed in the main part of this thesis in Chapter 8.

3. Structural Query Optimization

This section focuses on the structural optimization aspect. It starts with a review of the optimization steps of classical database systems and points out their relevance and extensions for DSMS with special consideration given to Multi-Query Optimization (MQO). Then, in Section 3.2, cost models are presented which allow to evaluate and to compare different query execution plans. Finally, Section 3.3 discusses concrete techniques for structural DSMS optimization along with the aspired optimization goals, and illustrates them with examples from existing data stream systems.

3.1. Optimization Steps

Aside from parsing and translating a query to an internal representation, the basic optimization steps–namely the *logical query optimization* and the *physical query optimization*–should be emphasized. Both originate from the database field ([Dat00]) but are of importance for data stream systems as well. Moreover–and in comparison to database systems–Multi-Query Optimization is a fundamental technique used for the execution of standing queries. The increased optimization potential of MQO has two reasons: First, a high number of standing queries is supposed to be evaluated in parallel, and second–due to the principle of a DSMS–the lifetime of a query is definitely longer than in traditional database environments. The query model discussion of Section 2.2 with the distinction between a logical and a physical query representation acts as the basis for these optimization steps. However, Multi-Query Optimization cannot be clearly identified as part of the physical or of the logical optimization step–it is involved in both of them.

3.1.1. Logical Query Optimization

Logical query optimization is based on heuristics, including techniques like early selection operations, early or invariant aggregation operations, or occasional early projections. Query expressions may be simplified and unnested, and common subexpressions can be recognized as well. This is topic of MQO and will be described later in this section. Explicit logical optimization should be performed by every DSMS, even though it is rarely mentioned explicitly. For example, from [Sta04], it is known that STREAM applies techniques like pushing down selections below joins, or eliminating redundant 'istream' operators as well as redundant projection operators.

A different approach is used by the NiagaraCQ system ([CDTW00]), where selections are pulled up and joins are pushed down to allow for maximum sharing of common subexpressions ([CDN02]).

3.1.2. Physical Query Optimization

The physical query optimization primarily considers the statistics of the data streams which are to be processed. The challenges at this point are manifold: In the style of database systems, different physical counterparts may exist for one logical operator. For example, join operators can be accelerated if one of their inputs is blocked ([HAE05]) or if the input stream rates are different ([KNV03]). In contrast to database systems, access path information, like indices, are not required for accessing the input data (stream) but for retrieving intermediately materialized data (stored in synopses) efficiently. Additionally–and this is very important for cost estimations–statistics like cardinalities do not make much sense in the stream context; new measures like stream data rates must be defined for the optimizer to work with. Furthermore, on the physical layer, operators performing complementary work may be combined to reduce the runtime overhead in terms of intermediate queues and computational resources.

3.1.3. Multi-Query Optimization for DSMS

The goal of Multi-Query Optimization is to save computational resources by re-using intermediate results. It is based on similarities among the standing queries concurrently present in the DSMS. According to the isolation principle of transactional query processing, a challenge for MQO is that the result of a single query must not be influenced by concurrent standing queries, not even if execution plans are shared. An overview of MQO for DSMS is given in [KS03].

General MQO approaches aim at identifying and re-using common parts of the query tree and are independent of the individual stream operators. Basic work was done in the NiagaraCQ system ([CDTW00]), where common parts of queries to XML data have been identified based on query signatures. Furthermore–and in contrast to the general optimization approach–selections may be pulled up in the query graph to enable more general query subexpressions near the leaves to be shared. Furthermore, dynamic regrouping is favored as new queries enter the DSMS and old queries are dropped.

The MQO concept of STREAM ([MWA$^+$03, ABB$^+$04]) is illustrated in Figure 3.1. Synopses (like $store1$ and $store2$) may be shared among multiple standing queries. So-called 'stubs' are used to facilitate the connection of multiple partial query graphs to a single synopsis (store) and thus the re-use of intermediate query results. In the example in Figure 3.1, the FIFO queues q_3 and q_4 feed the binary join operator's input windows which are materialized within the shared strores $store1$ and $store2$.

Aside from the general MQO approaches, some strategies focus on exploiting query similarities at the level of single operators like join or aggregation.

MQO Approaches for Aggregation

[ZKOS05] describe the hierarchical computation of aggregates. As shown in Figure 3.2, fine-grained aggregates $ABCD$ are computed first. Then, they are used by multiple queries to create coarser aggregates (grouping by ABC, ABD, BCD and AB, BC, BD, CD, respectively) in a hierarchical manner. A similar concept is applied during CUBE

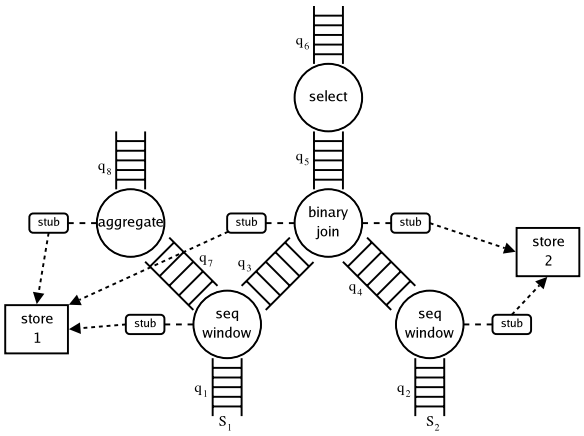


Figure 3.1.: Sharing Synopses in STREAM ([ABB+04])

computations [GBLP96, SAM98], except that only one query is involved which requires various grouping combinations.

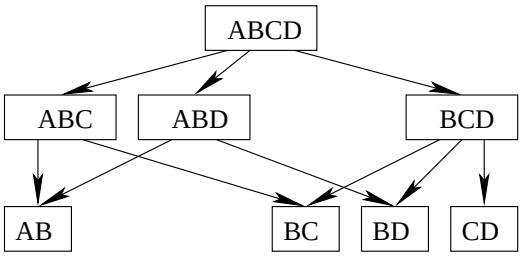


Figure 3.2.: Sharing Computational Efforts for Aggregation Operations ([ZKOS05])

MQO approaches for joins

Aside from aggregation, concurrent join operations can benefit from sharing resources, too. [HFAE03] propose to share join windows as illustrated in Figure 3.3. If the input signatures of two join operators are the same, only one physical window per join input (which is $W3$ in this case) is required. The windows $W1$ and $W2$ form the basis for the evaluation of two other joins which have the same input signature but vary in the window length specification. Different join evaluation techniques, such as largest window only (LWO), shortest window first (SWF) and maximum query throughput (MQT), may now be applied to produce join results for the tree queries. Using LWO, each new tuple is probed against all tuples of the largest window of the partner stream (the black tuples in the figure match the join predicate). Based on the timestamps of the tuples which contribute to a join result, the output tuple is routed to the appropriate query. To overcome the drawback of a delayed output for short windows, the SWF

strategy preferably probes new tuples against the shortest window, then against the next larger window, and so on. Due to the fact that probing in one of the larger windows is interrupted every time a new tuple arrives, the SWF strategy comes at the cost of delays for joins in larger windows. As a compromise, the MQT strategy does not unconditionally interrupt the probing procedure in larger windows if new tuples arrive: it additionally takes into account how many queries are likely to be served by the larger window probings and is thus more flexible.

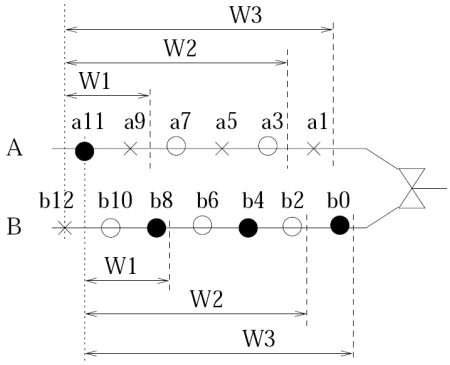


Figure 3.3.: Sharing Windows for Executing Different Join Operations ([HFAE03])

With their TelegraphCQ system,[CCD+03] and [MSHR02] propose a so-called CACQ (continuously adaptive, continuous query) approach, which is a modification of their eddies query processing framework (details in Section 5) and allows for the simultaneous execution of multiple queries. For multiplexing the intermediate query result, an extra state (tuple lineage, [CCD+03]) is maintained. With the concept of the 'grouped filter' operator, computational efforts for filters with similar predicates may be shared by using only one single operator. A grouped-filter index is maintained for each attribute of each standing query, which helps to efficiently compute overlapping portions of range queries by returning only those queries a tuple matches with (regarding the indexed attribute). To scale with the number of queries, multiple pipelined hash joins are optimized in a way that they share their index structures.

MQO results in different logical plan alternatives, which may originate from generic as well as from specific MQO approaches. On this basis, semantically equivalent QEPs are created and evaluated on the basis of a cost model, which is topic of the next section. The *naive approach* is to start with an initial plan, to apply all possible transformation rules and thus, to create a variety of plan alternatives (as presented in [RSSB00, CDN02]). After subsuming and comparing the costs of the individual plans, the 'best plan' is chosen for execution.

To reduce the optimization overhead, often only a limited number of transformations are applied which aim at reducing a certain cost measure or at increasing certain QoS metrics. The focus is on this *direct approach* whenever existing optimization techniques are presented in the remainder of this chapter.

The optimization approaches presented later in this section are supposed to work on the basis of a *query network*, which is the result of the Multi-Query Optimization.

3.2. Cost Models and Functions

This subsection reviews the cost models and resource calculation approaches proposed in the literature. It starts with considering cost models for specific operators and continues with models which allow for obtaining the costs of a whole operator graph.

The result of the cost model application can be either an abstract or a concrete measure of the required resources. The former may consist of 'tics,' 'timerons' or 'tuples' and only impacts the preference of a certain query execution plan over another. The latter contains the processing times of operators and the memory consumption of operators or queues specified in bytes. It additionally enables us to check available against required resources and thus, to perform some sort of admission control or resource reservation.

Generally, costs within a DSMS are based on data rates and on processing costs of single tuples, respectively, rather than on cardinalities of relations. The processing costs of a single tuple consist of costs for storing or filtering it, costs for probing them within a join opperation, etc. To determine the total costs required for a query plan, the per-tuple costs are multiplied by the data rates of the respective operators the tuples pass. The particular DSMS cost models differ in the per-tuple cost metrics and are denoted as *unit-time-based*.

3.2.1. Operator-Specific Cost Models

Most work towards operator-specific cost models has been done for the join operation, which is costly in database systems as well as in data stream systems. For example, [VN02] propose a data-rate-based cost model suitable to find the optimal join order within a query graph. The cost calculation in terms of processing time is based on the steps leading to the join result: costs of input projection and selection are considered together with the per-tuple handling cost for each of the join inputs. The cost model is adapted for cost estimation of different join implementations like nested loop join and symmetric hash join.

In [KNV03], a unit-time-based cost model is proposed, which–similar to the previous one–focuses on the cost of single join steps like probing, insertion and invalidation. Different cost calculation approaches are used for different scenarios; for example, limited CPU resources are used for join computations, or limited memory resources are used for holding join input windows. Furthermore, this unit-time-based cost model acts as the basis for the cost estimation of sliding window multi-joins in [GÖ03c].

Another unit-time-based cost model for join and filter operations is presented in [SMW05]. Based on the assumption that higher nodes in the distributed query tree have larger processing capabilities, their cost model for filter operations considers the filter's selectivity, the per-tuple costs of the filter scaled by a factor describing the operator level, and the network transmission costs between two nodes of the QEP. For join operations, the join selectivity as well as the per-tuple processing costs are considered.

3.2.2. Generic Cost Models

The generic cost models proposed here aim at covering the resource computation for arbitrary operators which are present in the query graph. When doing so, statistics like data rates, processing times and selectivities are taken into account.

The cost model proposed in [GV04] is used for optimizing QEPs, first, by creating the optimal dataflow plan regarding the data rates, and second, by reducing or eliminating conflicts regarding data handed over from operator to operator. Such conflicts (blocking) occur if a consumer operator has to wait for data from its predecessor operator. In detail, this unit-time-based cost model calculates the total processing time of a dataflow expression, which includes the time required for each operator as well as the operator selectivities.

[CKSV06] proposes a cost model which assumes stream rates to be constant on average. On the basis of stream characteristics, like the tuple validity interval lengths, the memory requirements of an operator's internal state can be calculated. [CcR+03] describe the cost of an operator graph with the measures 'processing costs per tuple' and 'box call overhead.' Due to the abstract character of the resource description, this simple cost model is only used for the qualitative comparison of QEPs. In [BBDM03], costs are considered in terms of generic operator selectivity and per-tuple execution costs of a certain operator. In analogy to [CcR+03], the resulting cost measure is an abstract number and therefore only suitable for comparing query plans or execution strategies.

The cost model of QStream favors the non-blocking data exchange and provides resources suitable for giving QoS guarantees. It is presented later in Section 8.1.

3.3. Structural Optimization Techniques

Goals like minimizing the resource consumption and maximizing the Quality-of-Service are targeted during the physical stage of the structural optimization. Often, it is difficult to distinguish between these two goals: If computational resources are saved by efficiently structuring the operator network, these resources are automatically available for increasing the processing quality and vice versa. The general approach is to maximize the result quality if the resources are limited and to minimize the resource consumption if a certain result quality is expected by the user. In the following, these two goals are considered independent from each other.

3.3.1. Minimizing Resource Consumption

In [CcR+03], the execution overhead is reduced by grouping operators together to create so-called 'superboxes.' Thereby, the intermediate queues are eliminated, which is beneficial for the memory consumption as well.

Most of the other strategies aim at minimizing the processing time of a standing query. For a single join operation, [KNV03] try to reduce the per-tuple costs for processing sliding window joins. They propose different join strategies for the case that resources like CPU or memory, or even both, are limited. Furthermore, they optimize the system

for different stream arrival rates of the two join input streams by using join algorithms (like hash, nested loop) asymmetrically. [HAE05] aim at reducing the blocking behavior of the window join in case of delayed arrivals by performing suitable join operations while waiting for certain input tuples and then buffering and ordering the join result. Their approach is implemented in the NILE system ([HMA+04]).

If more than one join operation is present, [BMM+04] aim at finding the optimal join order. They suppose many joins to be part of the query execution plan and use a greedy algorithm to find the globally optimal order in terms of minimal processing time. More general, [AN04] define a framework for static query optimization. Their goal is to minimize the processing time of a standing query containing operators with join or filter characteristics. The optimization is based on the processing time as well as on the selectivities of each of the operators. The cost of a join operation specifically contains detailed measures like the time required for window insertion or for probing tuples against windows.

The Gigascope DSMS ([CJSS03a]) optimizes query processing by breaking the query into high-level query nodes (HFTA–High Filtering, Transformation, and Aggregation) and low-level query nodes (LFTA–Low Filtering, Transformation, and Aggregation). The LFTAs accept network protocol data as input, whereas the HFTAs work on stream data. The optimization goal is to push as much query evaluation work as possible down to the network interface card (NIC) to reduce the stream load early by exploiting the NIC processing capabilities. More complex query evaluation work is then performed by the DSMS, where available resources (especially in terms of memory) are considerably higher.

3.3.2. Maximizing QoS

Maximizing Quality-of-Service incorporates the achievement of a high (output) data rate and the reduction of the query answer delay time as much as possible.

In [VN02], a function for describing each operator's impact on the data is derived from the costs for the single processing steps an operator has to perform. In their work, they include operators like selection, projection, Cartesian product and join. For the latter, specific implementations like nested-loop join or symmetric-hash join are considered. The optimization goal is either to maximize the amount of produced tuples for a certain point in time or to minimize the time required to produce a fixed amount of output tuples. Thus, their strategy–which is implemented in the NiagaraCQ DSMS–generally aims at maximizing the data rate.

Within the previous section, [AN04]'s work was mentioned as an approach to minimize query processing times. If resources are limited, it is also possible to put the optimization focus on maximizing the result data rate with the given (limited) resources. A maximum processing rate is also targeted by [GV04]. Out of various QEPs, the one with minimum processing time requirements (including waiting (blocking) times) can be chosen by applying their cost model. It is based on operator characteristics like selectivity and processing time.

For multiple join operations present in a query graph, [VNB03] propose an MJoin operator, which has the potential for maximizing the result data rate. [GÖ03c] advance in the same direction by using a multi-directional nested-loop join for n-ary join processing. The join evaluation may be done in an 'eager' or in a 'lazy' way, which results in lower or higher output delays and in more or less required resources, respectively.

3.4. Summary

Within this section, structural optimization was considered on the logical as well as on the physical level of the query representation. Depending on the application requirements, different optimization goals like resource minimization or Quality-of-Service maximization may be met by arranging operators in the optimal way. Furthermore, computational efforts in evaluating standing queries can often be shared. For adjacent application areas, many other optimization techniques exist.

The thesis refers to related optimization work on accessing stored relations on disk, including the swapping during operations ([UF00]), on the optimization of distributed data stream processing as in Borealis ([AAB+05, HBR+05, ABc+05]) or StreamGlobe ([SKK04, KSKR05]), and on aspects of sensor network optimization ([YG02, MFHH03, MFHH05]).

The cost models proposed so far all aim to obtain resource requirements and use them for optimization purposes. In contrast, an integrated cost model for obtaining concrete, operator- as well as stream-based resources is the precondition for providing QoS guarantees. Furthermore, the calculated resources must allow for being mapped to the basic allocation procedure provided by the operating system underneath. None of the existing cost model offers such flexibility, and therefore, the thesis proposes a new cost calculation approach for the use within QStream in Section 8.1.

4. Temporal Query Optimization

After a discussion on achieving optimization goals by efficient operator arrangement in Section 3, the focus is now shifted to optimization through changing the execution order of the operators within a QEP. For evaluating standing queries, each of the operators has to be executed repeatedly to continuously process parts of the incoming data stream. The execution order covers aspects like *when*, *how long* and *how often* each of the operators is executed.

The following subsection uses the term *scheduling* as a synonym for the temporal operator arrangement and presents some introductory remarks from a general point of view. Thereafter, Section 4.2 and Section 4.3 emphasize the difference between scheduling mechanisms and scheduling strategies. The latter includes a discussion of the specific scheduling goals of *minimizing resources*, *maximizing QoS*, and *guarantee QoS*.

4.1. Preliminaries

Scheduling is well known in the context of operating systems. Hence, the DSMS scheduling concepts should be seen in comparison to or as an extension of the operating system's scheduling approaches.

4.1.1. Operator States

The states of operators can be compared to the states of processes, even though the concept may be applied on a higher level. Section 4.2 discusses different possibilities of mapping operator states to process states.

If a standing query is admitted, a number of operators (representing the query graph) are created and are ready for execution. The operators are mapped to processes or threads of the operating system. Generally, multiple operators run in parallel.

The operating system together with a *scheduler component* is responsible for frequently switching among them to enable a pseudo-parallel execution. The scheduler work can be described using a simplified state diagram of process execution as in Figure 4.1.

After successful admission, the operator is *ready* for execution and thus considered by the scheduler. The scheduler decides when to let the operator perform its work (*running*) and when to interrupt the operator. This depends on the number of other operators in the ready state as well as on the applied scheduling strategy. If a running operator is blocked due to input or output operations, it is moved to the *waiting state*, which enables other ready operators to become running operators. After I/O completion (if the input data has been read or the output data has been written), the waiting operator is marked as ready and the whole cycle starts again. This procedure lasts as long as the standing

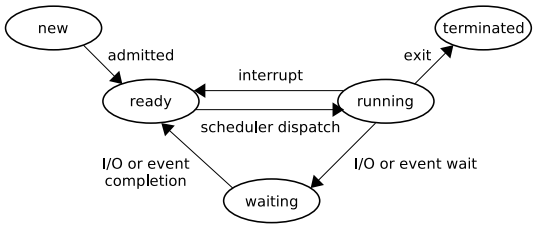
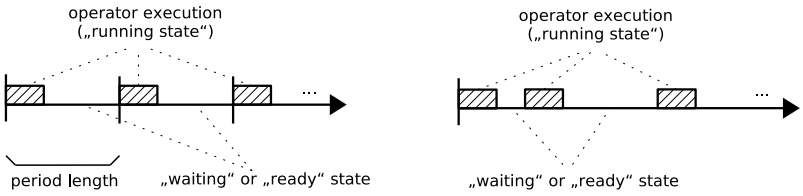


Figure 4.1.: Process State Transitions ([SGG02])

query is running. If the query is removed from the DSMS, its operators are *terminated* unless they are shared with other queries.

4.1.2. Continuously Running Operators

To continuously evaluate a standing query, each operator has to perform its work repeatedly as long as the standing query is present. Although, in general, a DSMS incorporates a push-based processing model, a distinction is made between the periodic and the aperiodic operator execution modes, which resemble pull-based and push-based processing models, respectively. The distribution of the operator work from the timeline perspective is illustrated in Figure 4.2. There, the shaded boxes represent the single *runs* of an operator. For sake of simplicity, other concurrently running operators are left out.



(a) periodic (b) aperiodic

Figure 4.2.: Operator Execution Modes

One way to handle repeated operator execution is to set up periods of work for each operator and to re-execute them periodically (*periodic operator execution*, Figure 4.2a). In this case, temporal query optimization has to ensure that each of the operators is executed or re-executed once within the defined period of time. Thus, in a broader sense, the periodic execution of operators resembles a pull-based execution model because the input data are 'fetched' from the stream sources. This requires intermediate buffers in between of every two operators. The operator execution frequency and the operator runtime within a single period should scale with the amount of input data.

In a push-based execution model, no periods are defined for the runs of the operators, and the operator execution must be directly triggered as the stream data arrive. Within such an *aperiodic operator execution* (Figure 4.2b), no explicit execution order is defined

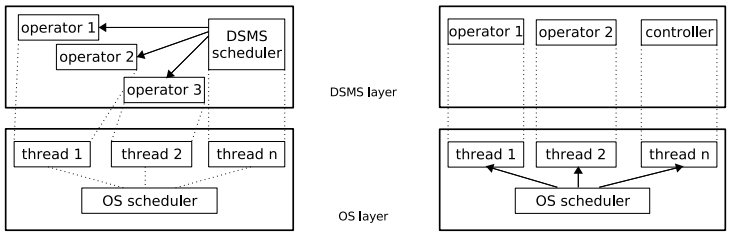
but–due to the incoming data–the time the operator spent in the running state should be proportional to its amount of work.

4.1.3. Scheduling Classification

Regarding the scheduling activities, a differentiation between *mechanism* and *strategy* is commonly accepted ([Tan92]). The scheduling mechanism describes how scheduling is implemented within the system. In contrast, the scheduling strategy determines the execution order of the individual processes. Scheduling mechanisms and strategies are topics of subsections 4.2 and 4.3, respectively.

4.2. Scheduling Mechanisms

The scheduling mechanism is concerned with the point where the scheduling decisions regarding the DSMS operators are made. On the one hand, scheduling can be controlled by an application program which is part of the DSMS (*DSMS level scheduling mechanism*). On the other hand, the scheduling actions may be handled entirely by the scheduler of the operating system (*OS level scheduling mechanism*). The mapping of DSMS operators to OS processes (Figure 4.3) is restricted by the applied scheduling mechanism as described below.



(a) DSMS level scheduling (b) OS level scheduling

Figure 4.3.: Scheduling Levels

4.2.1. DSMS Level Scheduling Mechanism

DSMS level scheduling (Figure 4.3a) means that one thread or process contains one or more DSMS operators. The DSMS scheduling decisions are entirely made by the *DSMS scheduler*; it decides which operator to run at what time by handing over the control to the operators. For example, the scheduler could directly call a function of an operator which–in turn–is responsible for performing one run. The work of the OS scheduler is not exploited by the DSMS and should therefore be invariant to the application.

Most of the current data stream systems (like Aurora [CcC+02], STREAM [ABB+03], PIPES [KS04], TelegraphCQ [CCD+03], Gigascope [CJSS03a]) incorporate the DSMS level scheduling mechanism. In Figure 4.4, the benefit of running multiple DSMS operators within a single OS thread within the Aurora DSMS ([CcC+02, CcR+03]) is

illustrated. The 'average latency' stands for the delay of output tuples using a specific system configuration that contains multiple standing queries. It is shown qualitatively that the runtime overhead using DSMS level scheduling can be reduced significantly because operating systems lack the facility for efficiently managing a large number of concurrent threads.

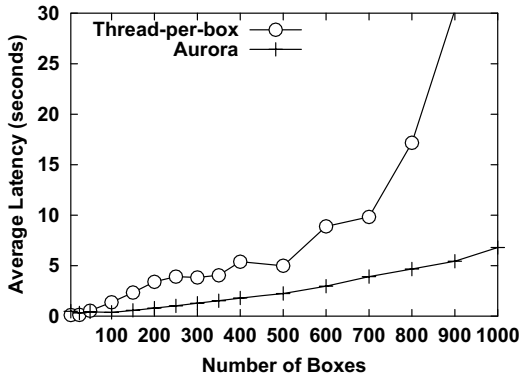


Figure 4.4.: Overhead of Thread Execution in Aurora ([CcR+03])

Within Aurora's so-called 'state-based execution model,' separate threads for the operators (worker threads) exist besides the scheduler thread. There, each worker thread is responsible for executing multiple operators. This results in higher flexibility at runtime due to the ability to group operators dynamically.

In comparison, only a single thread exists within the STREAM DSMS. Thus, no context switches are necessary on the operating system level for realizing the DSMS functionality; only procedure calls as high-level context switches are issued ([BBDM03]).

In the same manner, the Java implementation of TelegraphCQ ([SMFH01]) shares one thread among its operators (the eddy operator and the modules; more details on the TelegraphCQ architecture are given in Section 5.2). Furthermore, there exists an external thread pool for dispatching long-running system calls as well as specific threads for all user-defined operators.

Similar to the runtime architecture of Aurora, the PIPES system ([KS04]) can also adjust the execution overhead by mixing strategies from 'execute all operators in one thread' to 'execute each operator in its own thread.' This system incorporates a three-layer architecture to provide flexibility: On the first layer, operators are merged into a more complex node. On the second layer, one or more nodes are allocated to a thread. The third layer, finally, is responsible for scheduling the single threads.

4.2.2. OS Level Scheduling Mechanism and Optimizations

With OS level scheduling, a strict 1:1 mapping from operators to threads is applied. In comparison to DSMS level scheduling, the scheduling runtime activity is entirely handled by the operating system scheduler (Figure 4.3b). A controller component on the DSMS

layer is responsible for setting up and re-configuring the network as well as for initializing the OS scheduler with the user-defined parameters.

The benefits are, first, that the DSMS does not have to worry about scheduling mechanisms like switching over from one operator to another; it can focus on the scheduling strategy on top. Second, OS level scheduling allows for a very fine-grained time allocation ($< 1ms$) for the runs of the operators, which in turn allows for the negotiation of small output delays as Quality-of-Service requirements. The drawback of OS level scheduling is the high overhead at runtime due to the management of the large number of threads.

To reduce the scheduling or execution overhead of both DSMS and OS level scheduling mechanism, some optimizations have been proposed: On the one hand, operators can be batched or grouped like the superboxes of Aurora. The three-layer architecture of PIPES also targets this goal by being flexible in grouping operators on different levels.

On the other hand, tuples may be batched and thus processed during a single (larger) run of an operator. It leads to a lower overhead due to a smaller number of operator invocations and offers the potential for internal processing optimizations of the operator. This idea was proposed as *Train Scheduling* of the Aurora DSMS ([CcR+03]).

Furthermore, this thesis proposes a novel concept of batching input and output tuples, which is an integral part of the QoS guarantee DSMS concept. It is called *microperiods* and will be presented later in Section 8.2.3.

4.2.3. Scheduling Granularity

The scheduling levels outlined above focus on the layer on which the scheduler of the DSMS operators works. In addition, one may classify scheduling activities by the granularity at which the scheduler makes its decisions (Figure 4.5). The finest scheduling granularity is the single DSMS operator. A coarser granularity–like making scheduling decisions for operator groups or whole queries and applications–often comes with an inherent reduction of the scheduling overhead but goes at the cost of a fine-grained execution control.

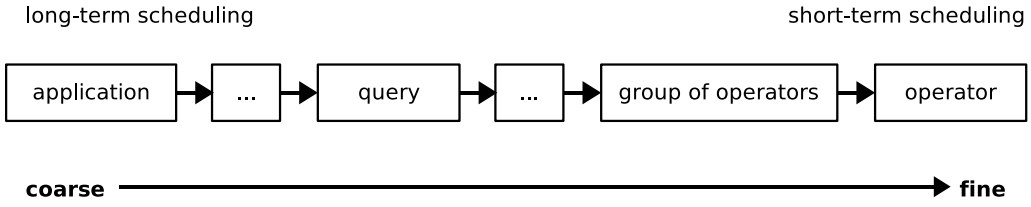


Figure 4.5.: Scheduling Granularity

Following the scheduling terminology from the operating system field, one can distinguish *short-term* and *long-term* scheduling. Some DSMS only support the finest scheduling granularity ([SLL05]); others support scheduling at multiple granularities (e.g. [CJSS03a, KS04]).

For example, Aurora's Application-at-a-Time (AAAT) strategy makes long-term scheduling decisions for the superboxes. Within a superbox, Box-at-a-Time (BAAT) scheduling (short-term) is used to determine the execution order of single operators.

Within STREAM, long-term scheduling refers to the operator chains, and short-term scheduling refers to the operators within a single chain. Following this concept, [JC04] use Path Capacity Scheduling to make long-term decisions for paths of operators. To the finer granularity of an individual operator (short-term), FIFO scheduling is applied in a bottom-up fashion from the last operator (of the path) to the first.

To summarize, scheduling of a DSMS cannot be treated independent from scheduling the operating system underneath. Different possibilities for the implementation of scheduling capabilities should be taken into account, which include periodicity, scheduling layer and scheduling scope.

4.3. Scheduling Strategies

The huge variety of applications which work on transient data requires a classification of the timely behavior of data stream processing systems. The goal is to ensure that the DSMS properties meet the requirements of applications and users. On the operating system layer, [SGG02] proposed three classes of application programs (interactive, batch, real-time) and derived scheduling criteria on that basis. In the same manner, the DSMS can be classified: An *interactive DSMS* requires a quick response to standing queries; so, the scheduling goal here is to minimize the response time. A *batch DSMS* would aim at processing as many data as possible within a given amount of time; thus, the focus is clearly on achieving a high data throughput. Last but not least, a *real-time DSMS* is supposed to guarantee certain QoS metrics like throughput or output delay. There, the maximization of single metrics is not that important.

As a result, each of the DSMS classes has specific requirements regarding the scheduling of their operators, which is determined by the concrete DSMS application.

To meet the specific scheduling requirements, a variety of scheduling strategies, which may be used for long-term as well as for short-term scheduling, have been established in the literature ([Tan92, SGG02]). Most of these strategies have been adopted for DSMS. An overview of the DSMS strategies along with their counterparts from the operating system field follows:

- **First-In-First-Out (FIFO)** focuses on the arriving tuples and feeds them through the operator network in arrival order.
- **Round Robin (RR)** iterates over the operators of the operator network and lets each of the operators work for a certain time quantum.
- **Greedy/Chain** preferably executes the most selective operators. It aims at achieving the minimum overall memory consumption for a given set of operators. Therefore, it relies on the operators' selectivities and risks the starvation of unselective operators. The comparable OS strategy is called Shortest Process First

(SPF) with the estimated process run times as scheduling criterion. The goal of SPF is a minimum waiting time for a given set of processes. In analogy to the Greedy algorithm, SPF bears the risk of the starvation of longer processes.

- **Priority-based scheduling** is a generalization of the three strategies mentioned above. The priority of an operator is determined by the user or by operator statistics like selectivity and arrival time. As a result, operators with a high priority are preferably scheduled.

More sophisticated scheduling strategies can easily be established, for example, in combination with different scheduling granularities. Furthermore, one can derive four criteria for evaluating and comparing DSMS scheduling strategies (Table 4.1): CPU utilization, memory usage, throughput and output delay. For sake of completeness, the corresponding criteria from the operating system field are also given in the table below.

OS Criteria	**DSMS Criteria**	topic of scheduling objective
CPU utilization keep the CPU as busy as possible; higher utilization is better	**CPU utilization** for a given number of queries, try to use as few CPU resources as possible; less is better	minimizing resource consumption, → Section 4.3.1
	Memory Usage for a given number of queries, try to use as little memory as possible; less is better	minimizing resource consumption, → Section 4.3.1
Throughput number of processes completed per time unit	**Throughput** amount of result data per time unit; higher is better	maximizing QoS, → Section 4.3.2
Response Time time until the process responds first	**Output delay** time the data need to pass through the network; smaller is better	maximizing QoS, → Section 4.3.2

Table 4.1.: DSMS Scheduling Criteria and Optimization Goals

The optimization following these criteria in terms of either minimizing or maximizing *CPU utilization*, *memory usage*, *throughput* or *Quality-of-Service* is topic of the following subsections.

4.3.1. Scheduling for Minimizing Resource Consumption

In comparison to the OS scheduling criteria, a DSMS scheduler has to consider the transferred volume of data. For example, within a DSMS, throughput is not a measure of the number of served processes but for the amount of processed data. Within this section, the various scheduling strategies are classified regarding the scheduling objectives.

Minimizing Memory Consumption

The memory consumption depends on the operator's queue length and the size of each operator's internal state. The former is required for handing over the tuples from one operator to another. The latter is determined by the number of tuples which the operator requires for performing its operation (join, aggregation, duplicate elimination, etc.). It can differ due to the appropriate algorithms and should be limited by incorporating window semantics; it was discussed in Section 2.2.2.

To minimize the queue sizes, the scheduler has to take the amount of transferred data into account. Each operator has a general selectivity assigned, which is based on DSMS statistics. It describes the relationship between the amount of input and output data during each processing step. Based on that selectivity, the scheduler calls the operator which reduces the amount of data most within the shortest time period. This strategy is implemented in Aurora and STREAM and is called *Min-Memory* [CcR+03] and *Greedy* [BBDM03], respectively. In Figure 4.6, the 'lower envelope simulation' of STREAM is shown. The circled corner marks (tuple states) denote the size of a tuple, which conceptually decreases during the processing. An operator execution changes the tuple state: it requires an amount of processing time (x-axis) and leads to further tuple size reduction (y-axis). The idea of the Greedy scheduling strategy is to schedule the operators with the steepest slope first.

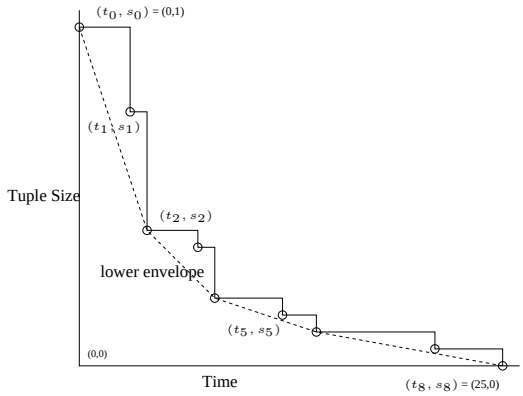


Figure 4.6.: Lower Envelope Dataflow Simulation ([BBDM03])

Further optimizations are produced by the *Chain* strategy, grouping adjacent operators together. The scheduling decision is then based on the coarser granularity of the chains. This is represented by the dotted lines in Figure 4.6. Thereby, Chain overcomes the problem of starvation which may occur with the Greedy approach, because Greedy does not take into account the data dependency between individual operators. Following the Greedy strategy, operators more downstream may receive a higher priority than operators near the data sources. As a result, the downstream operators may starve due to missing input.

A similar approach is presented by [JC04]. Their so-called *Segment Strategy* partitions an operator path. The partitioning criterion is the amount of memory which an operator is supposed to free up after a unit of work (memory release capacity). The segment construction along each operator path starts at the operators near the data sources and groups operators as long as the memory release capacity increases. Compared to the Greedy-like approaches, the Segment strategy is not the optimal approach with regard to memory consumption but achieves a smaller tuple delay. Furthermore, [JC04] propose a modified version of their scheduling strategy (*Simplified Segment Scheduling Strategy*), where only two segments per operator path are established. It is based on a user-defined threshold regarding the memory release capacity. The PIPES DSMS also implemented Segment Strategy scheduling ([CHK+03]).

In general, the Segment strategy as well as Greedy, Chain and Min-Memory rely on priority-based scheduling, where the priorities are defined internally following the potential memory reduction.

Minimizing CPU Utilization

The amount of processing time of an operator depends on the hardware, the applied algorithms, and the efficiency of the scheduling itself. The focus is put on the third issue and the minimization of CPU utilization is entirely associated with the reduction of the scheduling overhead by reducing the number of operator calls. This can be achieved by optimizing the scheduling mechanism (Section 4.2.2). Examples are Aurora's superboxes and tuple trains ([CcR+03]) as well as QStream's microperiod concept ([BSLH05]).

In addition, Aurora's *Min-Cost (MC)* strategy ([CcR+03]) aims at reducing the execution overhead by traversing the operator graph (of one superbox) in post order and by calling each operator exactly once during a cycle.

4.3.2. Scheduling for Maximizing Quality-of-Service

In the context of scheduling, maximum QoS metrics are associated with minimizing the *output delay* and with maximizing the *throughput*. Both issues are considered here.

Minimize Delay

Based on the FIFO strategy, the optimization goal within a DSMS may be to push or to pull a single tuple through the network of operators as fast as possible. Thereby, the response time is reduced and the initial output is produced early. The appropriate scheduling strategy (*FIFO*) used by the STREAM system ([BBDM03]) executes one operator after another, from the data source to the data sink.

Aurora's *Min-Latency* strategy ([CcR+03]) also aims at producing initial output as fast as possible. Output costs are assigned to each operator. This reflects the time which an operator and all its predecessors (of the same operator path) require to produce one output tuple. Thereby, operators with low output cost are preferably scheduled if tuples are waiting in their input queues. This increases the number of operator calls and thus the scheduling overhead.

The *Path Capacity Strategy* ([JC04]) picks the operator path with the largest processing capacity. Within the selected path, all tuples waiting in the input queue are processed one after another by the operators, from the data source to the data sink. The path processing capacity is a measure for the amount of tuples which can be fed through an operator path during one time unit. It is based on the operator processing capacity (tuples per time unit) and on the operator selectivities.

Maximize Throughput

If a scheduling strategy aims at minimizing the CPU utilization, it automatically enables a higher throughput with given resources. For example, in [CcR+03], Aurora optimizes the throughput by applying the Min-Cost strategy to reduce the scheduling overhead.

4.3.3. Scheduling for Guaranteed Quality-of-Service

For control and monitoring applications, it is important to require guarantees regarding the evaluation of standing queries. For example, the DSMS has to guarantee a minimum throughput of result data or it has to guarantee a maximum output delay as an upper bound for standing query evaluation.

The strategies of the previous section only focus on QoS optimization. There, the DSMS does its best to fulfill the QoS requirements but it may happen that–depending on the system's load–the QoS is lower than required. Even if the DSMS scheduling mechanism reacts quickly and the QoS guarantee is broken only for a short amount of time, this may not be tolerable for some applications.

To overcome this limitation, a DSMS can be designed to *guarantee* time-dependent QoS metrics. This is topic of QStream's different scheduling strategies, which are presented later in Section 4.3.

4.4. Summary

Scheduling is closely related to the concepts of the operating system underneath. For scheduling optimization, a distinction between 'mechanism' and 'strategy' is useful. The scheduling goal may be to either minimize resources, to maximize Quality-of-Service or even to give Quality-of-Service guarantees on time-dependent QoS metrics. The scheduling strategies incorporated by existing DSMS either aim at maximizing Quality-of-Service or minimizing the standing query's resource consumption. They are not adequate for providing QoS guarantees based on a respective user specification. Therefore, novel scheduling concepts for QoS guarantee DSMS are required; they are presented in Section 4.3 of this thesis.

5. Runtime Management

Within this section, different aspects of managing the DSMS during runtime will be discussed. The steps of the operational perspective are extended by runtime activities (Figure 5.1) which may be triggered by changes of the DSMS environment: Thereby, a clear distinction between *re-optimization* and *adaptation* of the DSMS is made. A re-optimization is caused by dynamically adding or removing standing queries, whereas the reason for an adaptation is changed operator statistics and data stream characteristics. Furthermore, an adaptation may only require a new temporal optimization, whereas a re-optimization mostly involves structural optimization as well.

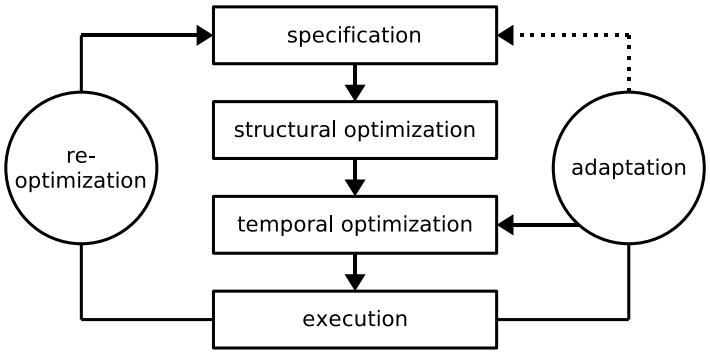


Figure 5.1.: DSMS Operational Perspective / Runtime Processes

Both runtime processes have influence on the resource consumption. Depending on the environmental changes, the DSMS may require either more or less resources for query evaluation. Generally, a lowered resource consumption is easier to manage than an increasing one because additional resources may not be available. The reason can be a lack of global resources or application-specific resource constraints. In both cases, the DSMS should continue working as efficiently as possible; thus, approximation techniques are used (Figure 5.2). Starting with a 'stable' system where sufficient resources for the query evaluation tasks are available, adaptation and re-optimization may lead to higher resource requirements and thus, to the application of approximation if the global resources are limited. All the runtime activities should make a 'stable' system again.

The rest of this section is structured as follows: First, Sections 5.1 and 5.2 give an overview of the re-optimization and adaptation techniques, respectively. Then, Section 5.3 summarizes current techniques of approximating the query result.

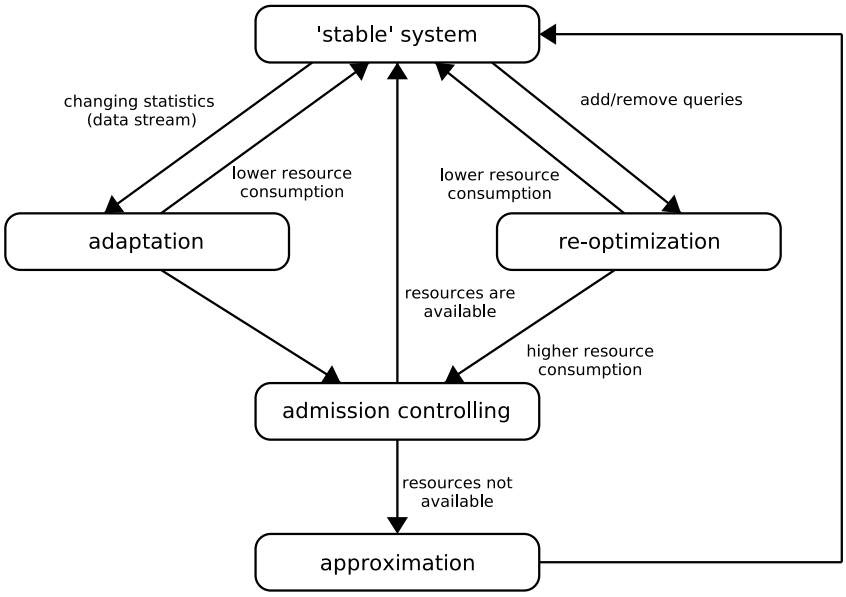


Figure 5.2.: Runtime Resource Management

5.1. Re-Optimization

Re-optimization must be applied if the set of standing queries changes. If no MQO was performed, the DSMS may simply add or remove queries with regard to resource limitations. With MQO, the situation becomes more complicated: On the one hand, if a query is selected to be removed, attention must be paid to those parts of the query graph which are used by other standing queries as well. Only the query graph which was solely used by the selected query can be dropped. On the other hand, a new query should be examined to use as many of the existing intermediate results as possible. In the following, three approaches are discussed which are directed towards re-optimization. The basis is data stream systems which are already capable of multi-query optimization.

First, the incremental query grouping technique of the NiagaraCQ system ([CDTW00]) allows the creation of groups of similar queries based on query signatures. If a new query is submitted and matches a certain group, the corresponding part of the query plan is replaced by the result of the identified group. If no existing group matches, the submitted query creates a new group. Dynamic regrouping, which includes merging existing groups if the workload has changed and queries were removed, is discussed in [CD02]. In [CDN02], a cost model for NiagaraCQ's incremental group optimization technique is proposed, which is based on characteristics like the number of queries, characteristics of queries, the distribution of distinct constant values, the update frequency and the update distribution. On that basis, the two alternative strategies of either pushing down or pulling up selections are evaluated.

Second, the CACQ (Continuously Adaptive Continuous Queries) concept of TelegraphCQ ([MF02]) offers some basic facilities for managing and optimizing the execution

of multiple standing queries but is restricted to Selection-Projection-Join (SPJ) queries. New queries may share the same scan operator if they address the same input data. Also, a single filter operator can be shared dynamically by adding multiple predicates to it. For sharing computational efforts at the join operator, new queries may be connected to existing stream indexes (which are implemented as State Modules (SteMs)) if the source stream matches.

The third approach is directed towards peer-2-peer networks. Their re-optimization aspect is similar to DSMS. For example, StreamGlobe ([SKK04, KSKR05]) focuses on adaptive query processing and optimization in streaming P2P environments. Therefore, the FluXQuery XML query engine ([KSSS04a, KSSS04b]) is used. A so-called 'Speaker Peer' is responsible for the optimization of a sub-network and for the coordination with the peer neighborhood. It continuously re-structures the network based on query commonality and load-balancing issues as well as on peer capabilities.

5.2. Adaptation

Based on changed characteristics of the data streams, it may be necessary to adapt the parameters of the running operator network. Both runtime activities–re-optimization and adaptation–take place independent of each other. The goal of the adaptation process is either *resource minimization* and *QoS maximization* (Section 5.2.1) or keeping the *QoS guarantees* (Section 5.2.2).

5.2.1. Adaptation for Resource Minimization and QoS Maximization

The most popular representative of data stream systems which incorporate adaptation is the TelegraphCQ system ([CCD+03]), which can be considered as an extension of the Telegraph system ([SMFH01]) to evaluate standing queries.

Regarding continuous adaptation, TelegraphCQ uses *eddies* ([AH00]) as its technical basis for implementing adaptive query plans. An eddy is an n-ary tuple router interposed between the data sources and the query processing operators. It reads the tuples from the data sources and routes them to the participating operators. When doing so, the eddy can adaptively change the routing to resemble different operator orderings. This is based on selectivities and costs of the operators as well as on the input data rates. The optimization goal is to favor adaptivity over best-case performance (due to the runtime overhead).

Figure 5.3 illustrates the interaction of eddies with 'State Modules' (SteMs) to resemble a join operator. A SteM is a temporary repository of tuples which supports the operations insert (build), search (probe) and delete. The input tuples are first sent as insert tuples to the first SteM and then as a probe tuple to the second SteM. The matching tuples returned by the SteMs are sent to the output. This provides flexibility and maximum query plan adaptation, as the eddy can route incoming tuples to either of the SteMs depending on their processing capability.

A list of *ready* and *done* bits within the eddies maintains the tuples' processing progress. Tuples obtain a low priority when they enter the eddy, and their priority

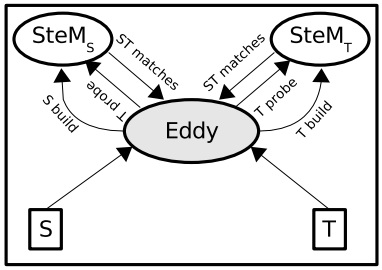


Figure 5.3.: Flexible Query Plans of TelegraphCQ ([CCD+03])

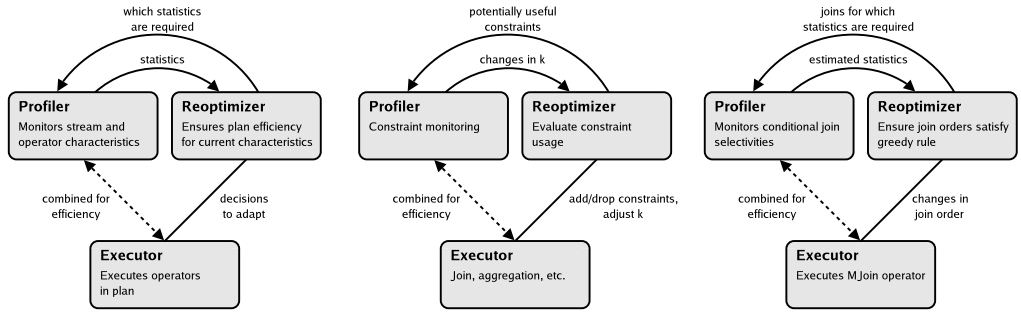
is increased each time they return from an operator. This ensures that tuples are processed completely, even if new tuples arrive. This can be compared with the *aging* concept of scheduling, which causes a stepwise decrease in the priority of processes to preserve low-prioritized processes from starvation.

The routing strategy of an eddy is based on the priority of the TelegraphCQ operators, which in turn is determined by each operator's consumption and production rate. Operators with higher efficiency at draining tuples from the system are preferred following the *Lottery Scheduling* ([WW94]). To quickly respond to dynamic fluctuations, a window approach is used for maintaining the statistics. This allows for more spontaneous reactions to changing input characteristic, as only the most recent statistics are considered. The operators have to 're-prove' themselves for each window.

In comparison to TelegraphCQ, the work of [ZRH04] is directed towards changing a formerly fixed query execution plan as statistics change. They propose two different strategies of migrating from one QEP to another if stateful operators like joins are involved. The challenge at this point is to consider tuples which are contained in the operator windows: The *moving state strategy* moves these tuples from the old QEP's operator states to the appropriate operator states of the new QEP. This causes a short delay at migration time. To avoid this, the *parallel track strategy* sets up the new QEP in parallel to the old one and runs both QEPs until the result tuples of the old QEP are not needed any longer (old states are 'expired'). Their solution is implemented in the CAPE DSMS ([LZJ+05, RDZ+05]).

Aside from the TelegraphCQ and the CAPE system, other DSMS incorporate adaptation without changing the query execution plan. [AN04] use a framework for determining the position of drop boxes triggered by continuously changing statistics. Within Aurora, the composition of superboxes can be changed dynamically based on the priorities of the operators, which are determined following current statistics and the QoS specification ([CcR+03]).

The STREAM DSMS uses *profiler* and *re-optimizer* components to monitor and to influence the query *executor*. Three different 'adaptation cycles' exist for optimization based on statistics like filter selectivities, k-constraints, and join selectivities from streams and operators, respectively (Figure 5.4). In each adaptation cycle, the profiler is responsible for collecting statistics. Then, the re-optimizer tries to find the optimal query execution and scheduling plan based on the optimization strategies of Sections



(a) StreaMon (b) k-Mon (c) A-Greedy

Figure 5.4.: Adaptation Cycles of STREAM ([ABB+04])

3 and 4. Finally, the executor puts the plan into effect. The runtime overhead is a trade-off between the speed and precision of the adaptation.

5.2.2. Adaptation for Time-Based QoS Guarantees

At runtime, specific attention must be paid to the operator network's behavior to permanently fulfill the QoS guarantees. The general problem is that the resource reservation in a QoS-guarantee DSMS is based on query and data stream statistics which were supposed to remain constant over time. Within realistic application scenarios, both assumptions do not hold. First, statistics of incoming data can only be estimated based on historical data and on experiences of the application administrator. Second, statistics will change over time as data stream characteristics change, too. Thus, the forecast of statistics becomes a challenging technique in the DSMS context. In particular, the adaptation for time-based QoS guarantees is presented in Chapter 9. It is part of the QStream robustness concept.

5.3. Approximation

Approximation techniques like sampling, window reduction and synopsis compression are required to lower the DSMS's resource consumption and to keep up with the arriving data. For example, queries can be evaluated with decreased quality, for example with regard to the precision or output data rate. The optimization goal at this point should be to *deliver best result quality* with the limited amount of resources. In order to do so, the administrator may define which quality metrics may be reduced without affecting the ability to still deliver valuable results.

Approximation may either be applied to the whole operator network, which means, for example, to sample the stream or to reduce window sizes, or it may target only specific (costly) operators for which the potential resource savings are maximal.

5.3.1. Generic Approximation

In [TcZ+03], the approximation strategy is based on dropping tuples within the query graph. They propose different placement strategies for so-called *drop boxes*. For only a single query, the drop box should be placed as far upstream as possible to reduce the stream load for all consecutive operators. If two or more queries share portions of the query plan, the drop box should be pushed as far upstream as possible until it reaches a connection point.

Figure 5.5 gives an example: If load shedding is only allowed for one of the connected queries Q_1 or Q_2, the drop box must reside downstream after the connection point B or D, respectively. If similar load shedding for both connected queries Q_1 and Q_2 is required, the drop box can be pushed further upstream to the connection point A.

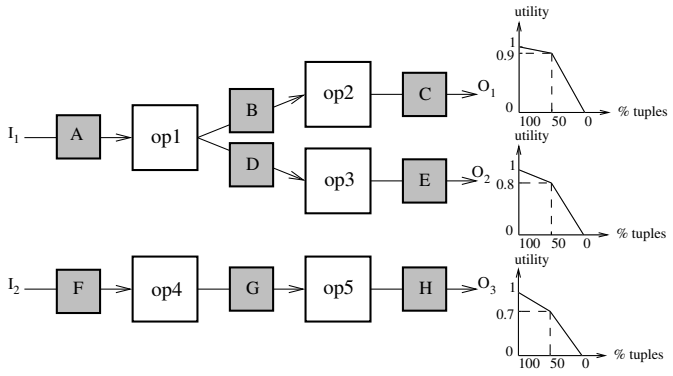


Figure 5.5.: Example Drop Box Placement ([TcZ+03])

A drop box may shed stream load based on three types of QoS specifications, as discussed in Section 2.2.2. In comparison to sampling approaches (like chain sampling or Bernoulli sampling [Haa05]), load-shedding decisions are not required to be of random nature. The general goal is to drop the tuples with the smallest benefit ('utility value') for the overall query result.

More general approximation techniques are proposed by [MWA+03]. They distinguish between *static* and *dynamic* approximation. The first technique is applied at the operator network's creation time and includes the reduction of window sizes for stateful operators as well as the initial reduction of the sampling rate. The second technique is applied at runtime and is thus suited for covering fluctuations in data rates, in the data distribution as well as in the query workload and the resource availability. In detail, on the one hand, the size of synopses (which keep the state of a stateful operator) may be reduced by incorporating sliding windows to limit the state or by only storing a sample of the synopses (histogram, compressed wavelet for aggregation and join; Bloom filter for duplicate elimination, set difference and intersection). On the other hand, additional sampling operators may be inserted into the query plan, or the sampling rate of existing operators may be reduced further. To summarize, STREAM offers a wide range of

different approximation techniques, whereas the Aurora DSMS is more flexible through its quality-driven load-shedding specification.

Aiming at adaptive memory management during query processing, [CKSV06] present an approach to reduce the window size during runtime. An initial window size is specified by the user (within the query) and is implemented as a window operator at the beginning of each path of the query plan. This determines the amount of tuples each subsequent stateful operator has to keep. In case of resource shortness, the window size is simply reduced.

The granularity of the tuples' timestamps additionally influences the memory consumption of the PIPES system: The PIPES aggregation algorithm is sensitive to time granularities, and thus, it has to store less aggregation values at a coarser time granularity.

Finally, all DSMS which support at least a sampling operator for accessing the data streams or for reading an analog sensor's value allow for approximation in terms of sampling rate reduction.

5.3.2. Operator-Specific Approximation

Aside from the general approximation approaches listed above, more specific load-reduction techniques can be implemented on the level of stateful operators.

Approximating Aggregation Operations

[BDM04] consider load shedding as an optimization problem of where and how much load to shed in order to achieve the optimal result in terms of maximum answer accuracy. Accuracy is measured as the deviation of the estimated answer produced by the system from the actual answer (without approximation). The concept of placing load shedders is similar to Aurora: if a standing query without shared segments is present, the optimal solution is to place the load-shedding operator in front of the first query operator. If an operator's output is shared among different queries, as shown in Figure 5.6, load shedders may be placed before and after operator B, which serves multiple queries. An exception is the query path which has the lowest sampling rate requirements ($P_4 = 0.5$) and thus the largest result quality q_{max}. In this path, only one load shedder is placed before operator B.

More specifically, [DGGR02] approximately answer aggregate queries by using sketch summaries if the available memory is insufficient. They are able to give guarantees on the approximation error.

Approximating Join Operations

Join operations are also subject to approximation processes if computational or memory resources are limited. [KNV03] propose different techniques to deliver the best result quality in terms of a maximum subset of the actual join result in case that either computational or memory resources–or both–are insufficient. They divide the join cost into

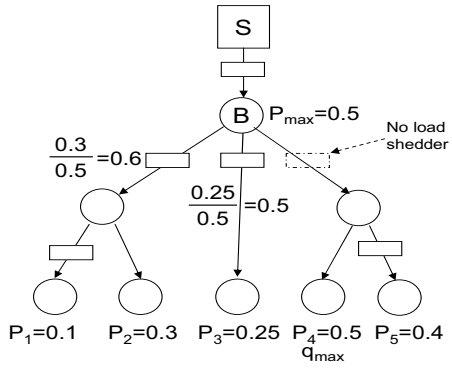


Figure 5.6.: Drop Box Placement in STREAM ([BDM04])

two independent terms, each corresponding to one join direction. This allows for graceful distribution of the limited resources depending on the input stream characteristics. First, if the join operator cannot keep up with the high stream arrival rates, they suggest to allocate the maximum amount of CPU resources to the join direction that evaluates the join from the smaller window to the larger one. Second, if not enough memory for holding both windows of the preferred size is available, most of the memory should be allocated to the window for the stream with the smaller input rate. Third, if both types of resources are insufficient, the largest subset of the result will be produced if the size of one window (memory resources) is maximized, with the effective input rate for the join partner (computational resources) being maximized, too.

[DGR03] also treat the size of the produced subset of join results as a quality measure which is to be maximized. They use semantic load shedding techniques to reduce the number of tuples buffered in input windows. Their tuple replacement strategy is priority-driven, where the priority correlates with the probability of a tuple to arrive in one of the input streams.

[SW04b] go one step further and propose two different approximation scenarios for sliding window join operators: Their approximation goal is either to deliver as many join results as possible (max-subset) or to create a random sample of the join result which may be more suitable for queries which involve an aggregation operation following the join. For both scenarios, an *age-based* stream model is supposed to be more suitable for many applications in comparison to the frequency-based stream model of the two former approaches: Using the age-based stream model, in case of memory shortness, not necessarily the oldest tuples within the join's sliding input windows are discarded. Instead, *age curves* indicate how likely a tuple is to produce join results if it becomes older; only tuples with a low utility value are discarded.

To summarize, approximation at a join operator (in terms of producing only a uniform sample as a subset) is often tolerated if an aggregation operation follows and uses the sample to provide a consistent and unbiased estimate of the true aggregate ([SW04b]).

5.4. Summary

Adaptation as well as re-optimization activities change the configuration of the DSMS. The challenges are the number and complexity of the standing queries and the characteristics of the incoming data streams–both change over time. If resources for query evaluation are insufficient, approximation techniques help reduce the workload to an acceptable level.

From the viewpoint of a QoS-guarantee DSMS, the adaptation as well as re-optimization procedure is of particular interest. The more often the DSMS configuration has to be changed (in either of the two ways), the more often the QoS guarantees are interrupted. This aspect did not receive adequate attention in the data stream community. For that reason, the QStream robustness concept was developed. It allows to trade the number of DSMS adaptations over time against the resources spent for standing query evaluation (Chapter 9).

Summary of Related Models and Systems

Existing DSMS focus on efficient and flexible processing of data streams. They are based on stream-oriented data and query models and provide a variety of optimization techniques which are dedicated to increase the content-based result quality and to decrease the required resources. Both optimization aspects are targeted at the query construction and query submission time as well as at runtime of the DSMS.

The optimization goal of providing guarantees especially for time-based QoS metrics has not been considered so far. Therefore, strategies and techniques completely dedicated to providing QoS guarantees have been developed and are presented within the main part of this thesis. This includes all DSMS aspects ranging from an adequate model for standing queries, QoS and data streams to scheduling strategies, appropriate optimization techniques and runtime management.

Part II.

QStream: Towards a Robust, Quality-of-Service Guarantee Data Stream Management System

6. QStream Modeling Aspects

Within this section, the terminology of *data streams*, *Quality-of-Service*, *operators* and *operator instances* is formalized for the QStream context. It goes beyond the scope of existing definitions, which–in most cases–define the data stream as a sequence of data items with only one schema associated. Contrary to other DSMS, QStream particularly pays attention to the evaluation of the attribute values over time. The DSMS tuples only play the role of 'representation points' of the input signal; with some additional knowledge of the input data characteristics, one can exploit attribute evaluation properties for enabling more powerful and more meaningful operations on the data streams, which leads to a more valuable result. The result in turn can be described qualitatively using novel Quality-of-Service (QoS) metrics. Section 6.1 starts with presenting a formal definition of data streams based on the classification in [SFL05]. Thereafter, Section 6.2 introduces QStream's QoS metrics, and finally, Section 6.3 describes the operator model based on the data stream formalization.

6.1. Data Stream Model

The QStream data stream model describes the data structures on which a DSMS performs its operations. Therefore, a *stream descriptor* is handed over from one operator to another.

Definition 6.1. *A stream descriptor $\hat{S}$ is a quadruple consisting of a data stream S, a schema E and a stream type C and the minimal time distance ΔT between two consecutive stream tuples.*

$$\hat{S} := (S, E, C, \Delta T)$$

A definition for the stream S, the schema E and the type C is given next. The minimum timestamp distance ΔT acts as an upper bound for the number of attribute value changes per time unit and is thus important for resource planning. Furthermore, a data stream S can be decomposed into partial streams which describe the evaluation of single attribute values over time. Orthogonally, the decomposition of a single stream tuple results in several attribute values.

6.1.1. Data Streams and Stream Tuples

To describe the layout of an individual stream tuple, the notion of a stream schema is introduced first.

Definition 6.2. *A tuple schema E is a sequence of attributes $E := (e_1, ..., e_i, ..., e_m)$.*

Each attribute $e_i \in E$ has a domain assigned, which is denoted as $dom(e_i)$ and defines the range of all possible attribute values along with an order relation.

Tuple Timestamps

In particular, the *time domain* $(M, <)$ used by QStream is a non-empty set of time instances M together with a total strict order $<$ (based on [BDE+97]). As a consequence, no duplicates regarding the tuple timestamps are allowed within QStream.

Definition 6.3. *A stream tuple T is a finite, ordered collection of attribute instance values a. It corresponds to a schema E and is defined as:*

$$T := (a_1, ..., a_i, ..., a_m)$$

There, m specifies the cardinality of the stream tuple T and the notation is $m = |T|$. The operation $\models_E$: $T \mapsto E$ denotes the affiliation of T to a certain schema E. The notion a_i $(1 \leq i \leq m)$ specifies the i-th attribute instance value of the stream tuple T. The first attribute instance value of a stream tuple T is always the tuple's timestamp:

$$\forall T : dom(e_1) = (M, <)$$

Definition 6.4. *A data stream S is a potentially infinite sequence of stream tuples T_j following the same schema E. An ascending order is defined on the timestamp attribute of the stream tuples:*

$$S := (T_1, ..., T_j, ..., T_n \mid \forall j(1 \leq j \leq n \wedge n \to \infty) : T_j \models_E E$$

The schema assignment can be extended to the whole stream: $S \models_E E$ denotes that the whole data stream S (all of its tuples) follows the schema E.

For simplicity, the stream may also be seen as a two-dimensional array with the first dimension describing the attributes of a stream tuple and the second dimension describing the consecutive tuples:

$$\begin{aligned} S &:= (T_1, ..., T_j, ..., T_n) \\ &:= ((a_1, ..., a_i, ..., a_m)_1, ..., (a_1, ..., a_i, ..., a_m)_j, ..., (a_1, ..., a_i, ..., a_m)_n) \\ &:= ((a_{1,1}, ..., a_{i,1}, ..., a_{m,1}), ..., (a_{1,j}, ..., a_{i,j}, ..., a_{m,j}), ..., (a_{1,n}, ..., a_{i,n}, ..., a_{m,n})) \end{aligned}$$

Consequently, $a_{i,j}$ specifies the i-th attribute instance value of the j-th stream tuple of stream S.

6.1.2. Partial Streams and Stream Classes

QStream allows for a description of the attribute value behavior over time. Therefore, first, the entity of a *partial stream* is introduced and second, different *partial stream classes* are proposed which can be assigned to the former.

Definition 6.5. *A partial stream is a stream of two tuples.*

$$S^P := S(T_1, ..., T_j, ..., T_n) \; with \; T_j(a_1, a_2)$$

A partial stream S^P is said to be contained in a stream $S (S^P \in S)$ if the tuples of S^P pairwise correspond to the tuples of S and each tuple of S^P is a projection on the timestamp attribute e_1 and on one further attribute e_i of stream S.

$$(S^P \in S) \quad \Leftrightarrow \quad S^P := \Pi_{e_1,e_i}(S) \quad with \quad S \models_E (e_1, ..., e_i, ..., e_m)$$

There, it is assumed that the semantics of the relational projection operator Π are known from the context of database systems so Π can be applied to stream tuples in a straightforward manner.

Going one step further, the notion

$$S[S_1^P, ..., S_i^P, ..., S_x^P]$$

denotes that all partial streams S_i^P make up the stream S. There, x equals the number of attributes of S decreased by one ($x = |S| - 1$) because each partial stream corresponds to a stream of attribute values. An exception is the first attribute of S - it is always the timestamp attribute. Orthogonal to the composition of partial streams, the notion

$$S(T_1, ..., T_j, ..., T_n)$$

expresses that a stream S contains a (possibly infinite) sequence of stream tuples T_j. The concept of partial streams is illustrated in Figure 6.1. The tuples $a_{1,1}, ..., a_{1,n}$ denote the timestamp of consecutive stream tuples. The other attributes form the partial streams $2, ..., m$.

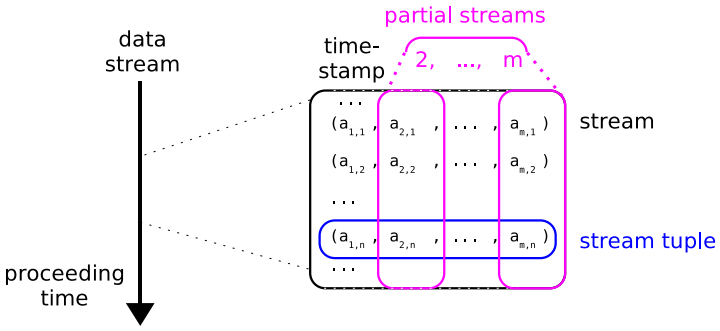
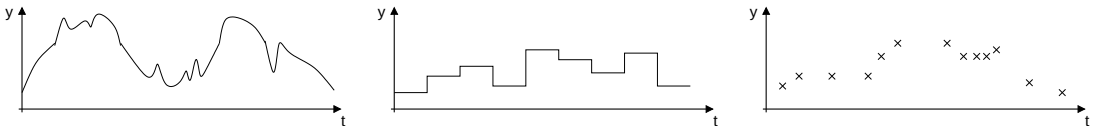


Figure 6.1.: Relationship between Stream, Partial Stream and Stream Tuple

The attribute evaluation characteristics are assigned as a *stream class* to each partial stream. Therefore, QStream relies on the stream classes which have been introduced in [SFL05]. The motivation is the presentation of real-world data by a sequence of DSMS

tuples. In some cases this does not sufficiently reflect the (behavior of the) input data, because the attribute values in between of two tuples might have a certain behavior–for example, it can be a constant or it may change smoothly.



a) continuous b) discontinuous c) event

Figure 6.2.: QStream Stream Classification

As shown in Figure 6.2, the partial data streams are classified with regard to the steadiness of their attributes. The stream tuples are considered to be *representation points* of the source signal (with a minimum time distance of ΔT). The attribute value behavior between two consecutive tuples is an important application characteristics and therefore, the following three partial stream classes are defined:

- **Continuous Partial Streams (CS):** A continuous partial stream, originating from a sensor measuring physical values such as temperature or pressure of a natural or an industrial process, shows uniformly continuous characteristics (Figure 6.2a). The sensor is supposed to output analog values which have to be discretized regarding time and quantized regarding their values to create the DSMS tuples (Figure 6.3).

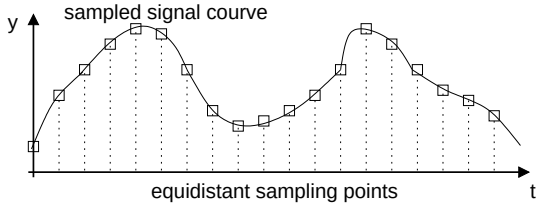


Figure 6.3.: Creating Tuples from the Sensor Signal

- **Discontinuous Partial Streams (DS):** A discontinuous partial stream represents data values which are constant during certain time intervals, e.g. aggregated data, like sum or average values (per day, month, year). There, data such as supermarket prices with daily updates, or stock exchange rates disseminated every hour, minute or second, impose regular attribute value changes and other streams–for example the placed bids of an auction–impose irregular attribute value changes with the exception that auction prices can only be increased.

- **Event Partial Streams (ES):** Event partial streams depend on sporadic real-world events and are only defined at the time of the event (as opposed to the former classes). Such streams consist, for example, of tuples coming from network traffic observations or from the monitoring of click-streams (Figure 6.2d).

Definition 6.6. *A partial stream class $c \in \{CD, DS, ES\}$ of a partial stream $S_i^P \in S$ describes the attribute evaluation characteristics of the $i+1$-th attribute of stream S, formally described as:*

$$S_i^P \models_c c \mid c \in \{CD, DS, ES\}$$

The first attribute of a stream is always the timestamp; thus, the i-th partial stream S_i^P corresponds to the $i+1$-th attribute of the stream S. The class of a partial stream is maintained as additional metadata. As a benefit of the above classification, stream elements may be processed using operators especially suited to appropriate partial stream classes. For example, the data rate of a continuous partial stream should only be reduced using a resample operation instead of applying traditional probabilistic sampling algorithms in order to keep the continuous characteristics and to reason about the bandwidth property which is typical for data streams and signals of analog origin (see Section 6.2 of the QoS model).

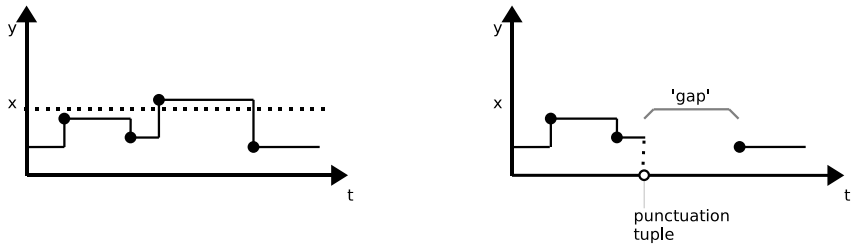
The partial stream classes c_i of partial streams S_i^P belonging to stream S are summarized as the stream type C.

Definition 6.7. *The stream type C summarizes the partial stream classes c_i of all partial streams S_i^P belonging to a stream S. The notation is:*

$$S \models_C C(c_1, ..., c_i, ..., c_m) \;\Leftrightarrow\; \forall S_i^P (1 \leq j \leq m) \in S \;:\; S_i^P \models_c c_i$$

6.1.3. Stream Punctuation

If a partial stream is continuous or discontinuous, it is assumed that the attribute value is defined even though no actual tuples arrive. Consequently, the most recent attribute value is assumed to stay valid until the next stream tuple arrives. Operators like join and aggregation are sensitive to the input stream classes. The problem at this point is that–if previous stream operations did not forward the stream due to its operation–so-called 'gaps' arise within the stream. Figure 6.4 illustrates an example where a filter operator only passed by stream tuples with a value smaller than x. The consecutive stream operators must be informed about the beginning of such a gap.



a) original discontinuous partial stream b) values above x deleted

Figure 6.4.: Necessity of Punctuation Messages

This is achieved by a *punctuation message* at the beginning of a gap. The punctuation message indicates that–although the attribute value is not changed explicitly by a new tuple–the partial stream attributes are not defined any longer.

Punctuation messages are explicitely required if discontinuous partial streams are involved: For continuous partial streams, equidistant tuple arrival is expected and thus an operator can 'automatically' detect a gap if no tuple arrives after a time of ΔT has elapsed. In the case of event streams, punctuation messages are not necessary, since an event partial stream is only defined at the points where tuples occur explicitly (not in between). Thus, each of the QStream operators has to consider punctuation messages during its operation and is supposed to insert a punctuation message into the output stream if a gap arises.

If gaps within the appropriate streams (partial streams) occur very frequently, the class of these partial streams can be considered as an event stream if the continuous or discontinuous characteristic is not given anymore.

Formally, a punctuation message is a tuple $T^p(ts, NULL, ..., NULL)$ which follows the current stream's schema. It contains the timestamp of the first deleted stream tuple and $NULL$ values for all other attributes.

6.2. QoS Model

One important design goal of QStream is to give Quality-of-Service guarantees. The implementation of that concept requires, first, the definition of QoS metrics suitable for data stream processing, second, the propagation of QoS metrics across all defined DSMS operators and third, the use of the result QoS metrics for QoS negotiation with the user.

The remainder of this section introduces the QoS negotiation concept and defines content- as well as time-based QoS metrics. The QoS propagation across the individual operators is described in Chapter 7, where the individual QStream operators are defined.

6.2.1. QoS Negotiation Concept

A variety of QoS metrics have been proposed for the use in data stream systems (related work, Section 2.3.1). The negotiation process is illustrated in Figure 6.5).

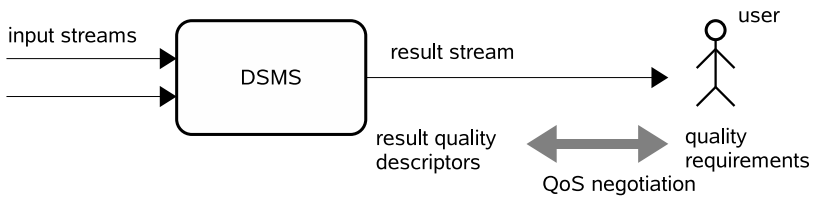


Figure 6.5.: QoS Negotiation Process

If the QoS delivered by the DSMS is equal or higher to the QoS requirement of the user, the negotiation is said to be successful. Otherwise, either the user reduces his QoS

requirements or the DSMS tries to trade the requirements against each other to come to an agreement.

The problem with well-established QoS metrics like 'sampling rate' ([MWA+03]) or 'aggregation precision' ([BSLH05]) is that it is nearly impossible to propagate them down the query graph through arbitrary operators and to include them in the QoS negotiation.

Therefore, it is absolutely required to establish novel QoS metrics which can be propagated through the query graph and thus also included into the QoS negotiation process; they are described in the remainder of this section.

6.2.2. Content-Based QoS Metrics

The content-based metrics may be assigned to data streams and describe the information they contain. We use the metrics *signal frequency* and *inconsistency* which are (a) novel QoS metrics in the DSMS context and (b) well-suited to describe the semantically enhanced data streams of the previous section.

Definition 6.8. *The* **signal frequency** F *is a measure for the amount of information contained in a continuous partial data stream. Only if all partial streams* S^P *of a data stream* S *are continuous (*$S^P \models_C CS$*), the signal frequency property can be assigned to the whole stream.*

The signal frequency stands for the highest frequency which may be contained within the partial continuous stream. The inverse of the minimum timestamp distance between two consecutive tuples, $\frac{1}{\Delta T}$, acts as an upper bound for F: Following the sampling theorem ([SW98]), it holds that $F \leq \frac{1}{2 \cdot \Delta T}$ with $\frac{1}{2 \cdot \Delta T}$ being the *bandwidth*; this term is commonly used in the signal processing area. The signal frequency of a stream can only be decreased during stream processing as no new information can be 'created.'

Based on a continuous partial stream S^P, the signal frequency F of data stream S is the minimum of all signal frequencies contained in the participating partial streams S^P belonging to S. The reason is that F represents a *lower bound* regarding the whole stream. It must be considered during the negotiation process, as users are supposed to always require a *minimum* signal frequency with regard to data stream S.

$$F_S = \min_{i=1}^{x} F_{S_i^P} \tag{6.1}$$

It would also be possible to assign a signal frequency to each participating partial stream and propagate them individually through the operator network. The drawback would be increased costs and complexity of maintaining the stream descriptors.

Definition 6.9. *The* **inconsistency** I *of a data stream describes the maximum deviation of the tuple's timestamp from the real-world event which is represented by the tuple's attribute values.*

An example is illustrated in Figure 6.6: A sensor delivers tuples of the form $T_1(t_1, a)$ and two other sensors deliver tuples like $T_2(t_2, b)$ and $T_3(t_3, c)$, respectively. These sensor

streams are partial streams S_1^P, S_2^P and S_3^P, as their tuples contain only one attribute value in addition to the tuple's timestamp. The inconsistency values I_1, I_2 and I_3 of each original sensor stream are supposed to be zero.

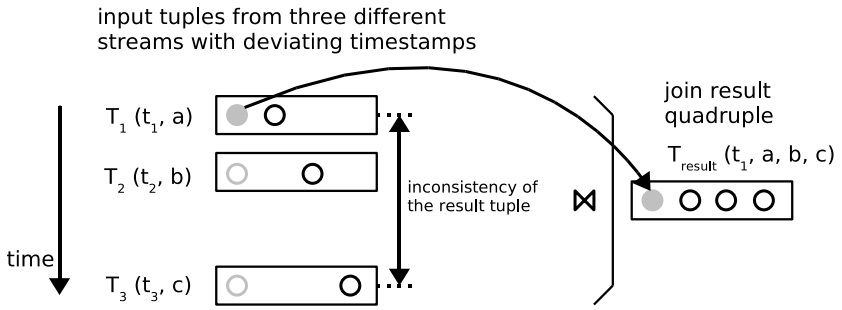


Figure 6.6.: Inconsistency at Join Operations

The goal is to merge the three sensor streams. A stream join would probe for join partners within a user-defined window and–if the input tuples match–it would output a quadruple like (t, a, b, c). Obviously, each result tuple can only have one timestamp, which is supposed to be $t = t_1$ in the example. The timestamp t does not exactly reflect the acquisition time of the attribute instance values b and c–the result quadruple $T_{result}(t_1, a, b, c)$ is inconsistent regarding time. The amount of inconsistency which is added to each of the partial streams contained in the join result depends in this case, first, on the join window size, and second, on the strategy of assigning the output timestamp.

In the example scenario, timestamp t_1 is used as the resulting timestamp and thus the inconsistency incurred by the join operation is $I_1 = 0$ for the first partial stream, $I_2 = abs(t_2 - t_1)$ for the second and $I_3 = abs(t_3 - t_1)$ for the third partial stream. The inconsistency either remains constant or increases during a stream operation. It is measured in units of time (e.g. in seconds).

Both content-based QoS metrics are summarized in a content quality descriptor.

Definition 6.10. *The content quality descriptor $\hat{Q}_{content}$ of a data stream S is a tuple containing the minimum signal frequency F as well as the maximum inconsistency I of all partial streams $S_i^P \in S$:*

$$\hat{Q}_{content} := (F, I \mid F = \min_{i=1}^{x}(F_{S_i^P}) \ \wedge \ I = \max_{i=1}^{x}(I_{S_i^P}))$$

If no signal frequency property can be assigned due to the occurrence of discontinuous or event partial stream classes, the value of F included in $\hat{Q}_{content}$ is simply $NULL$ and can therefore not be included into the quality negotiation.

An additional, but rather implicit goal of QoS-aware stream processing is to retain the stream classes of the partial streams. That is, keeping the characteristics of a continuous or discontinuous partial stream is supposed to be more valuable than simply changing the classes of all incoming partial streams to ES, although the latter would obviously decrease the processing complexity.

6.2.3. Time-Based QoS Metrics

In comparison to the content-based metrics, time-based QoS metrics refer to the runtime process of the query evaluation. In general, they address the speed of the query processing and consist of the **data rate** and the **delay** of a stream (of the result stream).

Definition 6.11. *The* **data rate** *R of a data stream S describes how many stream tuples per second occur in stream S.*

QStream works in 'real application time,' meaning that the timestamp distance between two stream tuples equals the elapsed time between the arrival of these two tuples at the DSMS. Thus, the data rate R a stream must be processed with is at maximum $R \leq \frac{1}{\Delta T}$. In case of equidistant tuple timestamps (continuous streams or regular discontinuous streams), a data rate of exactly $R = \frac{1}{\Delta T}$ is required to keep pace with the incoming data.

A stream will be delayed by each processing step independent from the data rate property: An operator reads a stream S, processes it and outputs a result stream S'. Thereby, the tuples formerly contained in stream S move to stream S' either in their original form or in a pre-processed form.

Definition 6.12. *The* **delay** *D of a stream S denotes the time distance between a stream tuple arriving at the DSMS and the insertion of this tuple into stream S'.*

The data rate as well as the delay are summarized as a time quality descriptor. It is assigned to a data stream descriptor $\hat{S}$ and refers to the runtime process of query evaluation.

Definition 6.13. *The* **time quality descriptor** *$\hat{Q}_{time}$ of a data stream S is a tuple containing the data rate R and the delay D of a data stream:*

$$\hat{Q}_{time} := (R, D)$$

6.2.4. Quality Request

The user may specify a quality request $Req(F_{min}, I_{max}, R_{min}, D_{max})$ containing time-based as well as content-based QoS metrics. The parameters refer to the query result stream descriptor $\hat{S}_{result}$ and have the following meaning:

- If the result stream entirely consists of continuous partial streams, it must be qualified for containing frequencies of F_{min} at least.
- The inconsistency of the result stream must not exceed a value of I_{max}.
- The result data has to be delivered at least with a data rate of R_{min} and
- The delay of the query result must not exceed the time D_{max}.

Thus, the query result quality ($\hat{Q}_{content}(F, I)$ and $\hat{Q}_{time}(R, D)$) has to meet the user's requirements $Req(F_{min}, I_{max}, R_{min}, D_{max})$:

- Req meets $\hat{Q}_{content} :\Leftrightarrow F \geq F_{min} \ \wedge I \leq I_{max}$ and
- Req meets $\hat{Q}_{time} :\Leftrightarrow R \geq R_{min} \ \wedge D \leq D_{max}$

6.3. Operator Model

Within this section, the QStream operator as well as the query model are presented. Thereafter, the concept of propagation of quality metrics is sketched.

QStream distinguishes between a specification and a runtime layer of operators and queries. Figure 6.7 illustrates that concept: on the specification layer, a set of *operators* is provided by the DSMS. *Standing queries* are specified by the user and composed of these operators. The runtime layer covers all aspects of standing query evaluation. If a standing query is to be executed, an instance of all its operators (*operator instances*) has to be created on a specific system. There, runtime properties are assigned. Finally, the operator instances make up the *standing query instance* which processes the data streams and delivers the result to the user.

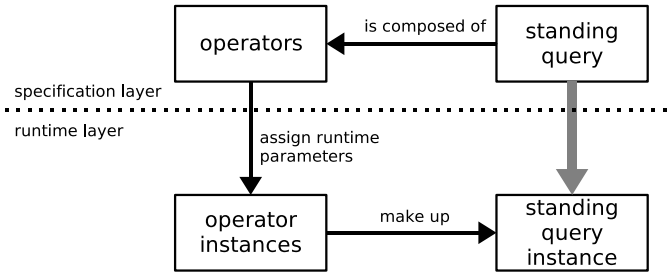


Figure 6.7.: Specification and Runtime Layer

6.3.1. Generic Operator Model

An operator is seen as the basic unit of work. Elementary operators are provided by the DSMS and enable the users to build their queries on that basis.

Operators

Definition 6.14. *An* **operator** $O(f_{\hat{S}}, f_{\hat{Q}_{content}}, \beta)$ *is a unit of work which processes either one or two input streams and results in one output stream. It is characterized by the two transfer functions* $f_{\hat{S}}$ *(functional description) and* $f_{\hat{Q}_{content}}$ *(non-functional description). Furthermore,* β *holds the set of operator-specific parameters.*

The transfer function

$$f_{\hat{S}} : (\hat{S}_1[, \hat{S}_2]) \mapsto \hat{S}'$$

states the operator's impact on the stream content, whereas

$$f_{\hat{Q}_{content}} : (\hat{Q}_{content,1}[, \hat{Q}_{content,2}]) \mapsto \hat{Q}'_{content}$$

describes the operator's influence on the content-based stream quality. The issues are illustrated in Figure 6.8. There, a source stream with the descriptor $\hat{S}$ and the content-based quality $\hat{Q}_{content}$ is processed by an operator O using the transfer functions $f_{\hat{S}}$ and $f_{\hat{Q}_{content}}$. The result is a modified stream descriptor $\hat{S}'$ and a reduced set of content-based QoS metrics $\hat{Q}_{content}$.

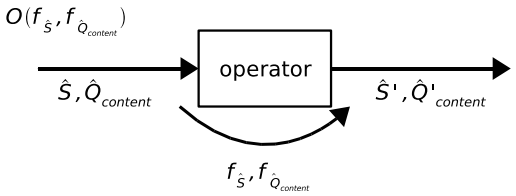


Figure 6.8.: Illustration of Operator Transfer Functions

Operator Instances

If an operator is to be executed within the DSMS, several operational parameters (like execution times, FIFO queue access characteristics, execution speed, etc.) are assigned.

Definition 6.15. *An* **operator instance** $OI(f_{\hat{S}}, f_{\hat{Q}_{content}}, f_{\hat{Q}_{time}}, \Phi, \beta)$ *is an operator* O *along with a transfer function* $f_{\hat{Q}_{time}}$ *of the time-based quality metrics and a set of operational parameters* Φ*.*

The transfer function

$$f_{\hat{Q}_{time}} : (\hat{Q}_{time,1}[, \hat{Q}_{time,2}]) \mapsto \hat{Q}'_{time}$$

describes how the operator instance influences the delay D and the data rate R of a stream. The operational parameters $\Phi((bi_1[, bi_2]), bo, t, P, s)$ extend the parameter set given in [BSLH05] and are mainly used for resource calculation and scheduling (Chapter 8). They refer to a single *run* of the operator instance and consist of

- the number of tuples *bi* (*batch input*) which an operator instance reads during each of its periods (during one buffer access)
- the average number of tuples *bo* (*batch output*) which the operator instance writes during each of its periods (during one buffer access),
- the operator's average processing time t per period,
- the period length P, and

- the size of the internal state (memory) s which an operator instance is allocated.

The ratio of *bo* and *bi* implicitly stands for an operator instance's 'selectivity.' It describes the data reduction between each input stream and output stream. For some operator instances, this quotient is a fixed value. For other operators, the average output size *bo* is data-dependent and thus subject of either estimation or statistics monitoring. Within Chapter 7, an operational description of all QStream operators is given. There, the necessity of a statistics-based selectivity value is stressed at the appropriate operators.

6.3.2. Standing Query Representation

The representation of a standing query is based on a directed acyclic graph (DAG) and relies on a set of elementary operators

Definition 6.16. *A* **standing query** Q *is a directed acyclic graph ([BBD+04, CcR+03, JC04]) with operators as nodes. The edges between every two nodes represent the data flow. An edge* $E_p < O_i, O_j >$ *indicates that the output data from operator* O_i *are taken as input for operator* O_j*. Thus, a query* Q *is described by a finite number of Operators* O_i *and a finite number of edges* E_p*:*

$$Q(\{O_1, ..., O_i, ..., O_x\}, \{E_1, ..., E_p, ..., E_y\})$$

The first operators are leaf nodes of the DAG. They have an incoming edge but no predecessor operator. In contrast, the output operator (root node) has an outgoing edge but no successor operator.

Definition 6.17. *An* **operator path** P *is a set of operators and edges which describe the data flow from one single leaf node (first operator) to the root node (last operator).*

Figure 6.9 illustrates an example of a standing query. There, two operator paths

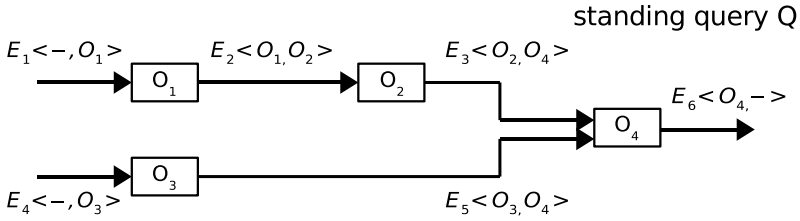


Figure 6.9.: Example Query Representation by a DAG

$P_1(\{O_1, O_2, O_4\}, \{E_1, E_2, E_3, E_6\})$ and $P_2(\{O_3, O_4\}, \{E_4, E_5, E_6\})$ are shown.

The concept of query specification can easily be transferred from the specification to the runtime layer by replacing all operators with their corresponding operator instances.

Definition 6.18. *A* **standing query instance** QI *is a standing query where all operators are replaced by the corresponding operator instances:*

$$QI(\{OI_1, ..., OI_x\}, \{E_1, ..., E_y\})$$

6.3.3. Quality Propagation

The negotiation of content-based QoS metrics can be performed on the basis of a standing query Q, whereas the negotiation for time-based QoS metrics can only be made for a standing query instance QI. The negotiation basis is the quality descriptors of the result stream (of the standing query and standing query instance, respectively). The resulting QoS descriptors have to be compared with the user's QoS requirements as described in Section 6.2.4.

Calculation of Content-Based Result Quality

The content-based quality $\hat{Q}_{content}(F_{source}, I_{source})$ of the data source(s) is given by the incoming data stream(s). There, the initial inconsistency value of I_{source} depends on the data acquisition strategy as well as on the timestamp granularity which was used by the data source (e.g. by the sensor device). For sake of simplicity and due to the fact that data acquisition is generally out of scope of DSMS processing, an initial inconsistency of $I_{source} = 0$ is assumed. The initial signal frequency value F_{source} depends on the sample frequency which was used while reading the analog sensor values and creating the continuous input stream.

The quality transfer functions $f_{\hat{Q}_{content}}$ of all operators are applied along the appropriate operator path from the leaf node operators to the root operator in order to calculate the content-based result quality.

$$\hat{Q}_{content,result}(F, I) = f_{\hat{Q}_{content,n}}(f_{\hat{Q}_{content,n-1}}(...f_{\hat{Q}_{content,1}}(F_{source}, I_{source})))$$

If a binary operator (which is the join operators in the case of QStream) is involved, its quality transfer function requires two quality descriptors of the appropriate input streams. Thus, the formalization is simply extended in that manner.

Calculation of Time-Based Result Quality

The time-based result quality consists of the result data rate R and an overall delay D. To determine R, the input data rate must be propagated across all operator instances using the implicitly given operator instance selectivity. Thereafter, the result data rate and the period length of all operator instances are available. The overall delay consists of the sum of all operator instances processing times plus the times required for data exchange. For obtaining both time-based QoS values, the QStream resource calculation has to be applied first. This will be described later in Section 8.

6.4. Summary

Within this chapter, the notion of data streams, Quality-of-Service, standing queries and elementary stream operators have been introduced. Each of the operators exerts influence on the data stream and its associated QoS properties. More specifically, on a specification layer of a standing query, the influence on content-based QoS metrics can

be described. Complementary, the propagation of time-based QoS metrics is subject to the runtime layer containing the standing query instance. Two issues are important regarding the QoS-aware data stream processing: First, the classification of data streams into different stream classes allows for specifically annotating QoS metrics depending on the application semantics. Second, the distinction between time-based and content-based QoS metrics points out QStream's efforts to close the gap in quality management, especially for time-based QoS metrics.

7. QStream Operators

This chapter addresses the individual QStream operators according to the formal introduction of operators. First, there are the helper operators $O_{resample}$ and $O_{reconstruct}$ (Section 7.1), second, there are the stateful operators $O_{aggregation}$, $O_{sync-join}$ and $O_{sampling}$ (Section 7.2) and third, the stateless operators O_{filter} and $O_{projection}$ (Section 7.3) complement the list. The focus lies on explaining the appropriate operators with a functional and a non-functional description. Furthermore, an operational description is given; it belongs to the operator instance OI and not to the operator O though.

- The **functional description** is represented by a transfer function for the stream content $f_{\hat{S}} : (\hat{S}_1[, \hat{S}_2]) \mapsto \hat{S}'$. It describes how the tuples of the result stream (descriptor) $\hat{S}'$ are created from the tuples of the input stream $\hat{S}$.
- The **non-functional description** consists of a transfer function $f_{\hat{Q}_{content}} : (\hat{Q}_{content,1}[, \hat{Q}_{content,2}]) \mapsto \hat{Q}'_{content}$ for the content-based quality in terms of signal frequency and inconsistency. A transfer function for time-based quality metrics is left out here, since it is topic of the resource calculation and scheduling sections.
- An **operational description** $\Phi(bi_1[, bi_2], bo, t, P, s)$ belonging to each operator instance resembles the operational parameters Φ which have been introduced in the previous section. Concrete values of period length P and of the average processing time t are not included in the operational description because these values are obtained from the JCP+ resource calculation and from statistics, respectively.

The only QStream operator which processes two input streams is the join operator. All other operators consume from a single stream and produce to a single stream. Therefore, for these unary operators, the functional, the non-functional, and the operational descriptions can be simplified as $f_{\hat{S}} : \hat{S} \mapsto \hat{S}'$, $f_{\hat{Q}_{content}} : \hat{Q}_{content} \mapsto \hat{Q}'_{content}$ and $\Phi(bi, bo, t, P, s)$.

Timestamp Semantics

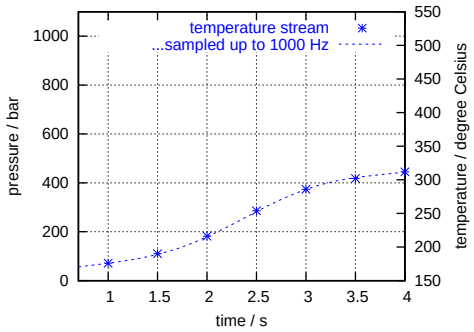
Different types of timestamps are of interest within QStream: The tuple timestamps (contained as the first attribute instance value) are considered to process the input data following the operators' semantics. They are used for the functional as well as for the non-functional operator description. In contrast, the tuples' arrival timestamps at the appropriate operator are decisive for the operational description. On this 'lower level,' the goal is always to provide sufficient input data for an operator and to always allow the operator to write its output data without being blocked.

7.1. Helper Operators

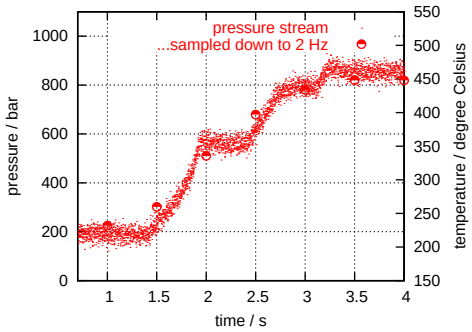
Two helper operators $O_{resample}$ and $O_{reconstruct}$ are used internally by other operators like join and aggregation. Helper operators are similar to ordinary operators. A functional, a non-functional and an operational description are included.

7.1.1. Resample

The resample operator applies downsampling as well as interpolation techniques to continuous partial streams. This allows the perfect reconstruction of intermediate tuples (interpolation, Figure 7.1 (a)) on the one hand and stream load reduction with a well-constrained loss of information (downsampling, Figure 7.1 (b)) on the other hand.



(a) Interpolating a continuous partial data stream



(b) Downsampling a continuous partial data stream

Figure 7.1.: Resample Operator Examples

Internally, the resample operator applies bandwidth-aware resampling, which was introduced in [SFL05] and must only be applied to partial streams S^P which are continuous ($S^P \models_c CS$).

The underlying concepts are *bandwidth-aware resampling techniques* [SG84]. They make possible the interpolation of tuple values between consecutive timestamps or the reduction of the amount of tuples of data streams with well-constrained loss of information content. Signal interpolation and digital filter design is a well-known domain and [SH90] is recommended for further information. Specific resampling techniques for digital signals are described in [SG84, Mat04].

The basis for this operation is a resampling factor $\frac{p}{q}(p, q \in \mathbb{N})$, where p is the interpolation factor and q is the downsampling factor. If $\frac{p}{q} < 1$, a low-pass filter has to be applied first to avoid aliasing effects. Depending on the required resampling factor, either p or q may be equal to 1 and thus, one of the sub-operations (interpolation, downsampling) can be left out.

The resampling operation uses a finite impulse response (FIR) low-pass filter representatively for many available digital filter algorithms ([SH90]). It has a cut-off frequency f_c for interpolating or bandlimiting a tuple stream. In both cases, a FIR filter kernel with

L coefficients of its transfer function is calculated. The cut-off-frequency f_c of the digital low-pass filter depends on the parameters p and q. Both resample steps are described in the following sections:

- **Bandlimited signal interpolation:** Interpolation is needed to reconstruct tuple values between tuples of consecutive timestamps. This is done in two steps. First, a number of $(p-1)$ so-called 'zero tuples' are padded between tuples $T_j(a_1, a_2)$ and $T_{j+1}(a_1', a_2')$. These are tuples whose attribute values (besides the timestamp attribute) are set to zero. Second, the zero-padded stream is convoluted with a filter kernel (cut-off frequency $f_c = p \cdot \frac{1}{\Delta T}$). The timestamps of the inserted tuples are distributed equidistantly between the timestamps of the tuples T_j and T_{j+1}.

  This results in a tuple stream with a smaller minimum distance of tuple timestamps $\Delta T'$ and exactly interpolated attribute values (Figure 7.1 (a)). The application of bandlimited signal interpolation is appropriate for handling signals of analog origin. This operation does not increase the stream's information content in terms of contained signal frequencies F.

- **Bandlimited downsampling:** Bandlimited downsampling is a combination of two steps: first, the tuple stream is bandlimited by applying a low-pass filter with the desired cut-off frequency f_c (again, a convolution with a filter kernel is applied), and second, only every q-th tuple (with its original timestamp) is passed by, since it is sufficient to represent the attribute value behavior of the bandlimited stream. Furthermore, the result tuples do not necessarily lie on the original signal curve, because the information content (the maximal frequencies which may occur) of the stream was reduced and only the trend of the signal remains after downsampling (Figure 7.1 (b)).

Both, interpolation and downsampling functionality are used by the resample operator. They are represented by the two functions and require a partial continuous stream S^P and the interpolation and downsampling factors p and q, respectively, as input.

$$S'^P = interpolate(S^P, p)$$

and

$$S'^P = downsample(S^P, q)$$

The resample helper operator instance $OI_{resample}(f_{\hat{S}}, f_{\hat{Q}_{content}}, \beta)$ has the input parameters $\beta = (p, q)$. The quotient of p and q stands for the resample factor. If $\frac{p}{q} > 1$, the amount of stream tuples is increased by that factor. Otherwise, if $\frac{p}{q} < 1$, the amount of stream tuples is reduced, which may come along with a reduction of the signal frequencies F contained in the stream.

Functional Description

The functional description of the resample operator is given as

$$f_{\hat{S}}(\hat{S}) \quad := \quad (S', E, C, \Delta T') \mid f_{\hat{S},S}(S) = S' \ , \ f_{\hat{S},\Delta T}(\Delta T) = \Delta T'$$

where $f_{\hat{S},S}$ describes the data stream transformation and $f_{\hat{S},\Delta T}$ denotes the transformation of the minimum timestamp difference. The schema E and the stream type C remain unchanged.

Result stream: The stream content transformation function $f_{\hat{S},S} : S \mapsto S'$ transforms a continuous partial stream S^P contained in S into another continuous partial stream ${S'}^P$. A description is given as follows:

$$f_{\hat{S},S}(S, p, q) \quad := \quad [{S'}^P] \mid {S'}^P = \mathit{downsample} \ (\ \mathit{interpolate}(S^P, p), q)$$

Here, *downsample* and *interpolate* are the two elementary signal processing functions described above. Both are supposed to work on the basis of continuous partial streams and also deliver a continuous partial stream as result. If a punctuation message arrives from the input stream, both interpolation and downsample do not output result tuples any longer. The punctuation message is forwarded. After the gap has occurred, a number of L consecutive stream tuples are required before the first results will become available.

Result minimum timestamp distance: The new minimum timestamp distance $\Delta T'$ directly scales with the inverse of the resampling factor $\frac{p}{q}$:

$$f_{\hat{S},\Delta T}(\Delta T, p, q) \quad := \quad \Delta T \cdot \frac{q}{p}$$

Non-functional Description

To propagate the QoS metrics properly, a non-functional description is given accordingly. The stream operators which internally use *resample* may also make use of these functions to 'propagate' the (partial) stream's quality.

Signal frequency F: The resampling factor $\frac{p}{q}$ may influence the frequency of each of the partial input streams. If $\frac{p}{q} > 1$, only interpolated tuples are inserted into the data stream but no higher frequencies (no new information) are added. F remains unchanged in this case. Otherwise, if $\frac{p}{q} < 1$, the stream's frequency F is decreased in case that the new minimum timestamp difference $\Delta T'$ is not sufficient to represent F.

$$F' := \min\left(F, \frac{p}{2 \cdot \Delta T \cdot q}\right)$$

Inconsistency I: The interpolation step does not increase the stream's inconsistency value as only new tuples are created and their inconsistency value is assumed to be zero. In comparison, during the downsampling step, a number of formerly independent stream attribute values become represented by one new attribute value. Thereby, the new inconsistency value is the overlaid time span of the inconsistency values of a number of q tuples.

Figure 7.2 illustrates the inconsistency propagation for a resampling factor of $\frac{p}{q} = \frac{2}{3}$. The input tuples are T_1, T_2 and T_3. Their timestamps constantly differ by an amount of $\frac{1}{F}$ with F denoting the input stream's signal frequency. The tuples' inconsistency interval is annotated above the time axis in gray color. In a first step, the interpolation function (with $p = 2$) creates one new tuple with a timestamp between every two existing tuples. The inconsistency intervals of the new tuples are generally not known; they are assumed to be zero. After the interpolation step, the tuples' timestamps differ only by a value of $\frac{1}{p \cdot F}$ and are equidistant, too.

Now, within a second step, a number of $q = 3$ tuples are 'merged' into one result tuple. The result tuple's timestamp can be determined arbitrarily from the timestamps of the merged tuples; the only requirement is that the result tuples' timestamps must be equidistant again. The result stream's inconsistency is at two times the former stream's inconsistency value (inconsistency of the first and the last merged tuple: $2 \cdot I$) plus the time distance between these two tuples, which can be expressed as $\left((q-1) \cdot \frac{\Delta T}{p}\right)$. The

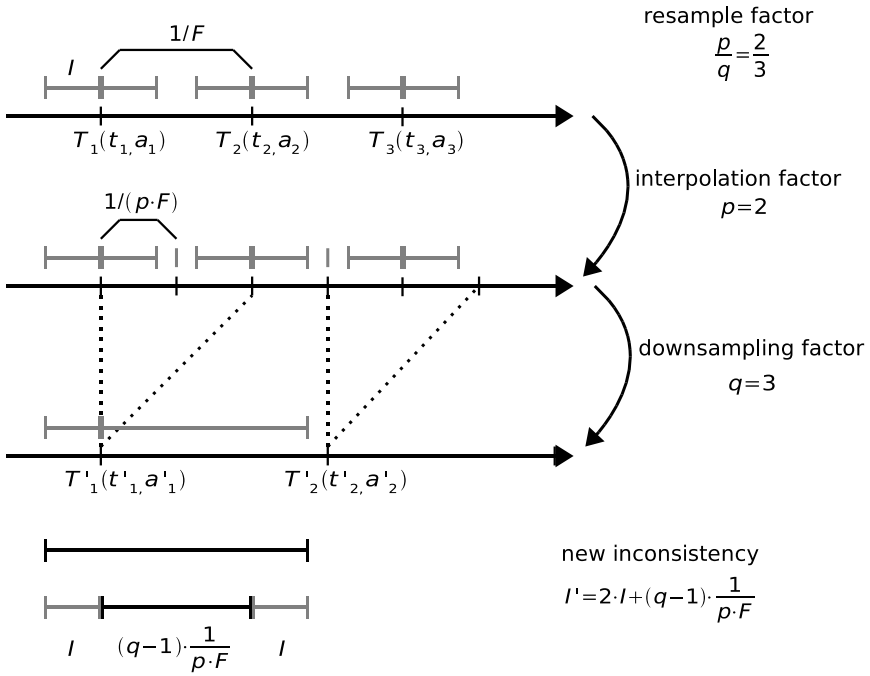


Figure 7.2.: Example of Resampling Operator Inconsistency Propagation

upper bound of the final inconsistency value can be calculated as

$$I' := 2 \cdot I + \frac{(q-1) \cdot \Delta T}{p} \tag{7.1}$$

There are certain special cases regarding the resampling factors p and q, which allow to define a closer upper bound for the final inconsistency:

- $q = 1$: Only the interpolation step is required. The formerly existing tuples are passed through together with some new tuples. The overall (partial) stream inconsistency remains unchanged as interpolated tuples are supposed to have an inconsistency of zero and the stream's inconsistency is the maximum inconsistency across all stream tuples: $I' = I$.
- $q \neq p + 1$: For that specific case, during the downsampling step, at least one of the merged tuples is a newly created one and thus has an inconsistency of $I = 0$ assigned. The resulting inconsistency is reduced to $I' = I + \frac{(q-1) \cdot \Delta T}{p}$ in this case.

Operational Description

From an operational point of view, the resample operator reads the input stream tuple by tuple and outputs on average a number of $\frac{p}{q}$ tuples during each processing step. It is a stateful operator which has to buffer a number of $2 \cdot L$ tuples, where L is the number of filter coefficients. The memory consumption is set to $2 \cdot L$ because a number of L filter coefficients as well as L stream tuples must be hold. Within QStream's implementation, L is assumed to be constant. An alternative would be to acquire L as a parameter provided by the user.

$$\begin{aligned} \Phi &:= (bi, bo, t, P, s) \\ &:= \left(1, \frac{p}{q}, t, P, 2 \cdot L\right) \end{aligned}$$

7.1.2. Reconstruct

The helper operator $O_{reconstruct}(f_{\hat{S}}, f_{\hat{Q}_{content}}, \beta)$ exclusively works on discontinuous streams DS. As input parameters, it takes a sequence of timestamps together with the associated minimum timestamp difference: $\beta = (SeqTS, \Delta T_{SeqTS})$. As illustrated in Figure 7.3, it creates intermediate tuples on the basis of the timestamps given by the sequence $SeqTS$. The attribute value of such an intermediate tuple is the same as the attribute value of the last arrived tuple.

Functional Description

The content transformation function $f_{\hat{S},S}$ consists of adding tuples to the stream and assigning a new minimum timestamp difference $f_{\hat{S},\Delta T}$:

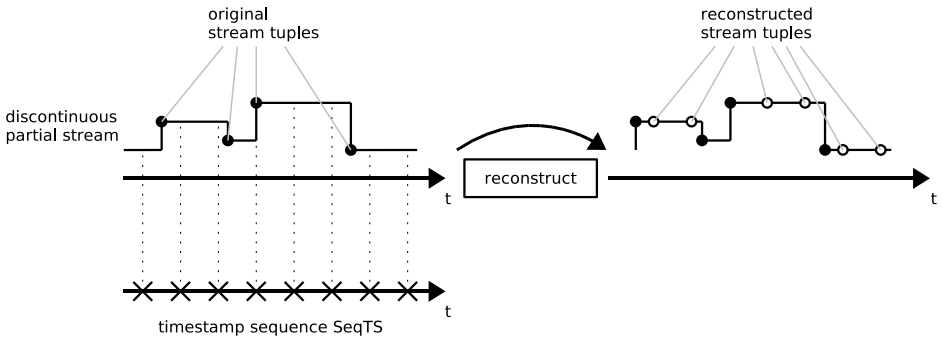


Figure 7.3.: Tuple Reconstruction within a Discontinuous Partial Stream

$$f_{\hat{S}}(\hat{S}, SeqTS) \quad := \quad (S', E, C, \Delta T') \mid f_{\hat{S},S}(S) = S' \ , \ f_{\hat{S},\Delta T}(\Delta T) = \Delta T'$$

The schema E and the stream type C are not changed.

Result stream: The stream content transformation function $f_{\hat{S},S} : S \mapsto S'$ transforms a discontinuous partial stream S^P contained in S into another discontinuous partial stream S'^P, no matter if S^P is of regular or irregular nature. A description is given as follows:

$$\begin{aligned} f_{\hat{S},S} \quad & (S(T_1, ..., T_j, ..., T_n), SeqTS(t_1, ..., t_u, ..., t_v)) := [S'^P \mid \\ & S'^P = (T_1, ..., T_u, ..., T_v \mid \forall u(1 \leq u \leq v) : \\ & T_u = (t_u, a_{2,j} \mid (a_{1,j}) \leq t_u < a_{1,j+1}) \wedge T_j \neq T^P)] \end{aligned}$$

Each of the timestamps t_u of the result stream tuples T_u corresponds to a given timestamp of $SeqTS$. The attribute value $a_{2,j}$ of T_u equals the attribute value of the last stream tuple $(a_{1,j}, a_{2,j})$ which is still valid at time t_u (due to the discontinuous character). The timestamps of the 'vertices' of the discontinuous partial stream are contained in $SeqTS$ and thus the characteristic stream behavior does not change.

If a punctuation message occurs, tuples are neither produced nor forwarded for the duration of the gap (except for the punctuation message itself).

Result minimum timestamp distance: The new minimum timestamp distance $\Delta T'$ is equal to the minimum timestamp distance of $SeqTS$, is provided by the calling operator as an additional input parameter, and is thus passed through:

$$f_{\hat{S},\Delta T}(\Delta T) \quad := \quad \Delta T_{SeqTS}$$

Non-Functional Description

The reconstruct operator only creates additional tuples. Their inconsistency values are set to zero; thus, the inconsistency I is not changed at this point.

$$I' = I$$

Furthermore, the signal frequency F is not considered because the reconstruct operator does not handle continuous partial streams.

Operational Description

The reconstruction operator reads the input stream tuple by tuple ($bi = 1$). The number of created intermediate tuples, and thus the average output batch size bo, strongly depends on the content of $SeqTS$ and is assumed to be given in statistical terms. Furthermore–as the reconstruction operator is supposed to work in pipeline mode–it has to keep only one 'regular' stream tuple as well as one tuple of the timestamp sequence. The timestamp sequence tuples will arrive more frequently than regular stream tuples. Every time a timestamp sequence tuple is available, its timestamp is 'paired' with the attribute value of the last regular stream tuple. Thus, an internal memory of two tuples is sufficient.

$$\begin{aligned} \Phi &:= (bi, bo, t, P, s) \\ &:= (1, bo_{reconstruct}, t, P, 2) \end{aligned}$$

7.2. Stateful Operators

Within the QStream system, the stateful operators *aggregation* (O_{agg}), *join* ($O_{sync-join}$), and *sampling* ($O_{sampling}$) are considered. They are described in detail in the following three subsections.

7.2.1. Aggregation

The aggregation operator $O_{agg}(f_{\hat{S}}, f_{\hat{Q}_{content}}, \beta)$ allows the computation of different aggregates of a data stream S. The operator-specific parameters are the length of the aggregation window t_{win}, a data rate reduction factor f and an aggregation function for each partial stream contained in S:

$$\beta = (t_{win}, f, (f_{agg,1}, ..., f_{agg,x}) \in \{AVG, SUM, COUNT, MIN, MAX\})$$

The aggregation operator entirely works on the basis of the tuples' timestamps. A so-called re-evaluation period $t_{re-eval}$ is implicitly determined by the result data rate requirement R_{min} (assume that R_{min} directly refers to the output data rate of O_{agg}) as $t_{re-eval} = \frac{1}{R_{min}}$. Based on the window length and the data rate reduction factor,

the QStream aggregation operator is restricted to resemble either *sliding* or *jumping* aggregation windows. In each case, the aggregate is computed every $\frac{1}{R_{min}}$ time units.

If a factor of $f = 1$ was selected, a sliding window is applied. The window length t_{win} determines which historical time span is to be involved in the aggregation.

If a factor of $f > 1$ was selected, the aggregation operation incorporates a jumping window. The window length t_{win} is fixed and implicitly determined by the result data rate: $t_{win} = t_{re-eval} = \frac{1}{R_{min}}$. The window is non-overlapping and there are no gaps regarding time. The factor f determines the required input data rate ($R_{input} = f \cdot R_{min}$) and thus the number of input tuples which arrive during the time span of t_{win}. f can be chosen arbitrarily by the user.

Functional Description

The aggregation operation is formally described as

$$f_{\hat{S}}(\hat{S}) \quad := \quad (S', E', C', \Delta T') \mid f_{\hat{S},S}(S) = S' \ , \ f_{\hat{S},E}(E) = E' \ , \ f_{\hat{S},C}(C) = C' \ , \ f_{\hat{S},\Delta T} = \Delta T'$$

where $f_{\hat{S},S}$ describes the data stream content transformation and $f_{\hat{S},E}$ and $f_{\hat{S},C}$ denote the transformation of the stream's schema and type, respectively.

Result stream: The aggregation result is computed individually for each partial stream. This is illustrated by the aggregation functions $f_{agg,1}, ..., f_{agg,x}$ in Figure 7.4.

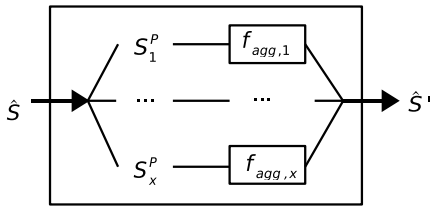


Figure 7.4.: Functionality of the Aggregation Operator

The appropriate aggregation function is applied to stream tuples within a window of length t_{win}. The maximum number of tuples contained in t_{win} depends on the stream's minimum timestamp distance ΔT and can be calculated as $n = \frac{t_{win}}{\Delta T}$. The concept is illustrated in Figure 7.5. Per aggregation group, one result tuple with the end timestamp of the window is produced.

In general, no aggregate value is produced during the time of a gap, even if the gap overlaps with the aggregation window. The punctuation message is forwarded to the result stream and the first aggregate values become available at the time where the gap does not overlap with the aggregation window any longer.

The transformation function $f_{\hat{S},S}$ takes a stream $S[S_1^P, ..., S_j^P, ..., S_x^P]$ as input and produces an output stream $S'[S'^P_1, ..., S_j^{P\prime}, ..., S'^P_x]$. When doing so, each partial stream S_j^P is aggregated separately using the appropriate aggregation function $f_{agg,j}$.

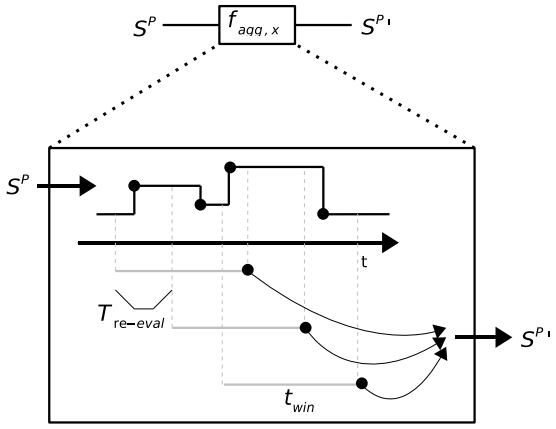


Figure 7.5.: Sliding Window Example of the Aggregation Operator

$$f_{\hat{S},S} \quad (S, T_{eval}, t_{win}, (f_{agg,1}, ..., f_{agg,j}, ..., f_{agg,x})) := [S_1^{P\prime}, ..., S_j^{P\prime}, ..., S_x^{P\prime}|$$
$$\forall j(1 \leq j \leq x, x = |S| - 1) : S_j^{P\prime} = f_{agg,j}(S_j^P, T_{re-eval}, t_{win}) \wedge S_j^P \in S]$$

The restriction within QStream is that not all of the aggregation functions ($f_{agg} \in AVG, SUM, COUNT, MIN, MAX$) are applicable to each partial stream class. The following distinction is made:

- **Continuous partial streams:** Here, MIN, MAX and $COUNT$ can be applied in a straightforward manner on the basis of the individual tuples contained in the interval. One must consider, however, that all three aggregation functions do not keep the continuous partial stream's characteristics. Instead, they result in a discontinuous stream where the aggregate value is valid for the duration of a re-evaluation period $T_{re-eval}$ (independently of sliding or jumping window aggregation).

  The SUM aggregate is not allowed as it would not reflect any useful semantics regarding the attribute values of analog (sensor) origin.

  The standard AVG aggregation function is not available for use either, since it would 'smoothen' the partial stream indefinitely without keeping the continuous characteristics. Instead, the resample helper operator is proposed to smoothen the attribute values, to reduce their cardinality, and to reduce the continuous partial stream's information content in terms of the signal frequency:

  $$AGG(S_j^P, T_{re-eval}, t_{win}) = resample(S_j^P, p, q) = resample(S_j^P, \Delta T, T_{re-eval})$$

  The resampling factor is $\frac{T_{re-eval}}{\Delta T}$, where $T_{re-eval}$ equals the new (minimum) distance between the timestamps of two consecutive stream tuples. The window

length t_{win} is not used for the resample operation; the number of required tuples for resampling is implicitly determined by the resample operator and its accuracy property in terms of the number of filter coefficients. Moreover, for the AVG aggregation of a continuous partial stream, the resample operator implicitly applies a sliding window in each case.

- **Discontinuous partial streams:** The standard MIN and MAX aggregation functions can be used for determining the time window's minimum and maximum attribute value of a discontinuous partial stream. SUM and $COUNT$ should only be used if the application context provides proper semantics for the appropriate results. The AVG aggregation is not allowed to work on the basis of the individual tuples contained in the aggregation window. Instead, AVG must consider the validity intervals of attribute values within the aggregation window. If the example data stream of Figure 7.6 shall be aggregated within the time span of t_{win}, three different intervals I_1, I_2 and I_3 must be considered. The time point t_{now} stands for the current time at which the aggregate is to be computed.

  The appropriate attribute values must be weighted with the length of their validity intervals. The length of I_1 is determined by the time difference between the window border and the first tuple arriving within the aggregation window. I_2 equals the time distance between the first and the second tuple of t_{win}. Finally, I_3 starts at the arrival time of the second input tuple but ends at the window border.

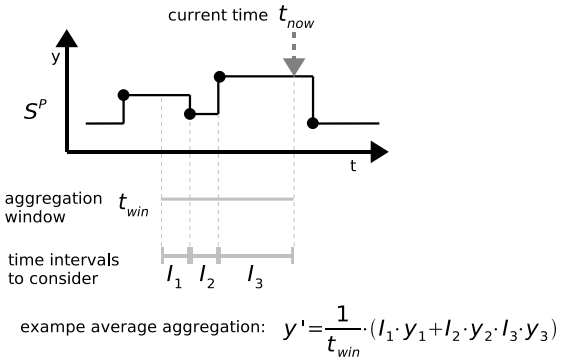


Figure 7.6.: Example for the Aggregation of a Discontinuous Partial Data Stream

To summarize, the aggregation operator considers both the set of tuples M ($|M| = n$) contained in the aggregation window t_{win} and the previously arrived tuple:

$$M(T_{i-1}, T_i, ..., T_{i+n})$$

The AVG aggregate is computed by cumulating all tuple attribute values $a_{2,x}$ (weighted by their validity time span, which in general is $a_{1,x} - a_{1,x-1}$) and dividing the result by the window length t_{win}. One must pay particular attention to the first

and the last time interval, both of which are limited by the aggregation window borders. They have a length of $a_{1,i} - (t_{now} - t_{win})$ and $t_{now} - a_{1,i+n}$ respectively.

$$AGG(S_j^P, T_{re-eval}, t_{win}) = \frac{1}{t_{win}} \cdot (a_{2,i-1} \cdot (a_{1,i} - (t_{now} - t_{win})) + a_i \cdot (a_{1,i+1} - a_{1,i}) + ... + a_{i+n} \cdot (t_{now} - a_{1,i+n}))$$

The aggregate values are produced with a timestamp distance of $T_{re-eval}$ and thus are equidistant. This is similar to the previous aggregation operations on discontinuous partial streams.

- **Event partial streams:** For event partial streams ($S_j^P \models_c ES$), all aggregation functions can be applied by default. The result stream changes to a discontinuous stream, as each aggregate value is valid until the next tuple's timestamp.

Result schema: If individual partial streams shall be excluded from the aggregation operation, the value of the appropriate aggregation functions must be given as $NULL$ ($F_{agg,j} = NULL$). The stream's schema E changes in a way that partial streams whose aggregation function is $NULL$ are not processed and therefore not part of the result stream.

$$f_{\hat{S},E}(E(e_1, ..., e_j, ...e_x)) = (e'_1, ..., e'_j, ...e'_x | e'_j = \left\{ \begin{array}{ll} \emptyset & \text{if } f_{agg,j} = NULL \\ e_j & \text{otherwise} \end{array} \right)$$

Result stream type: The stream type is changed depending on the applied aggregation function f_{agg}. Table 7.1 summarizes the stream class transformations.

class c_i of S_i^P	**aggregation function**	**class $c_i{}'$ of $S_i^{P'}$**
CS	AGG ($\rightarrow$ $resample$)	CS
CS	MIN, MAX	DS
DS	all	DS
ES	all	DS

Table 7.1.: Stream Class Transformations of the Aggregation Operator

If a continuous partial stream is AVG-aggregated (resampled), its continuous characteristics remain. If other aggregation functions are applied to continuous partial streams, the stream class changes to DS. Discontinuous and event partial streams change their stream class to DS in each case due to the repeatedly applied aggregation and the fixed re-evaluation period $T_{re-eval}$.

$$f_{\hat{S},C} \ (\ C(c_1, ..., c_j, ..., c_x)) = \left(c_1{}', ..., c_j{}', ..., c_x{}' \mid c_j{}' = \left\{ \begin{array}{ll} CS & \text{if } c_j = CS \text{ and } f_{agg} = AVG \\ DS & \text{otherwise} \end{array} \right) \right.$$

Result minimum timestamp distance: The result streams' minimum timestamp distance $\Delta T'$ equals the re-evaluation period $T_{re-eval}$:

$$f_{\hat{S},\Delta T}(\Delta T) = T_{re-eval}$$

Non-Functional Description

Signal frequency F: For a continuous partial stream, an AVG aggregation may decrease the stream frequency as only one aggregate value is produced at re-evaluation time. Thus, the signal frequency F' which is contained in the result stream is oriented towards the signal frequency propagation of the resample operator: It is the minimum of the former signal frequency F and the inverse of the half of the new minimum timestamp difference $\Delta T'$.

$$F' = \min(F, \frac{1}{2 \cdot T_{re-eval}}) = \min(F, \frac{1}{2 \cdot \Delta T'})$$

Inconsistency I: The inconsistency propagation is treated separately for continuous partial streams on the one hand and for discontinuous and event partial streams on the other hand.

If a continuous partial stream is AVG-aggregated, it directly follows equation 7.1 of Section 7.1. The resulting inconsistency value can be obtained as:

$$I'_{S^P} = 2 \cdot I + \frac{(q-1) \cdot \Delta T}{p}$$

and equals the inconsistency propagation of the resample operator, where $\frac{p}{q}$ is given as the pruned fraction $\frac{\Delta T}{\Delta T'}$. The aggregation window length t_{win} does not have any influence on the inconsistency propagation in that case.

The inconsistency propagation of a discontinuous or an event partial stream is also oriented towards the resample operator. The timestamps of the created tuples (aggregate values) may have a maximum inconsistency of the aggregation time span t_{win} plus two times the former inconsistency value–under the assumption that stream tuples have been present at exactly the beginning and at the end of the aggregation time span (upper bound). This is illustrated in Figure 7.7, where tuples T_1, T_2 and T_3 are present within the aggregation window of length t_{win}. The inconsistency interval is therefore calculated as:

$$I'_{S^P} = 2 \cdot I + t_{win}$$

The resulting inconsistency of the whole stream S' is the maximum of the resulting inconsistencies I_{S^P} of the aggregated partial streams. It is independent of whether or not a sliding or a jumping window aggregation has been applied:

$$I' = max(I_{S_1^P}, ..., I_{S_j^P}, ..., I_{S_x^P})$$

where

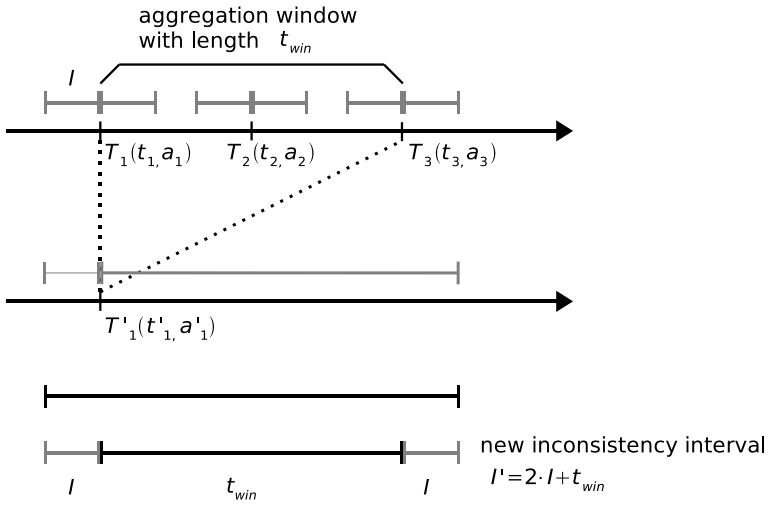


Figure 7.7.: Inconsistency Propagation of Aggregation Operator

$$I_{S_j^P} = 2 \cdot I + \begin{cases} \frac{(q-1)\cdot\Delta T}{p} & \text{if } S_j^P \models_C CS \text{ and } f_{agg} = AVG \\ t_{win} & \text{otherwise} \end{cases}$$

Operational Description

For the AVG aggregation of continuous partial streams, the operational description of the resample operator is used. The input stream is consumed tuple by tuple ($bi = 1$), where on average, a number of $bo = \frac{\Delta T}{\Delta T'}$ tuples are produced as output. Again, the internal state depends on the (constant and predefined) number of filter coefficients L.

$$\begin{aligned} \Phi &= ((bi, bo, t, P, s) \\ &= (1, \frac{p}{q}, t, 2 \cdot L) \\ &= (1, \frac{\Delta T}{\Delta T'}, t, 2 \cdot L) \end{aligned}$$

For other kinds of aggregations, different operational descriptions for handling sliding and jumping windows must be given. In both modes, the input stream is read tuple by tuple ($bi = 1$). The tuples are internally stored; therefore, a maximum internal memory of $s = \frac{t_{win}}{\Delta T}$ is required. For jumping window aggregation, the aggregate could also be computed incrementally without storing the individual tuples which belong to an aggregation window.

The average number of output tuples produced during each run depends on the length of the aggregation window t_{win} and on the re-evaluation period $T_{re-eval}$. Note that the applied re-evaluation period $t_{re-eval}$ can only be an integer multiple of an operator period

P due to the QStream operator implementation as periodically running components: $t_{re-eval} := \left\lceil \frac{t_{re-eval}}{P} \right\rceil \cdot P$. Thus, the smallest re-evaluation period is the period length of the aggregation operator, which in turn is calculated from the data rate requirements (Chapter 8).

- **Sliding window aggregation:** Here, one output tuple is produced per operator run ($bo = 1$).

$$\begin{aligned} \Phi &:= (bi, bo, t, P, s) \\ &:= \left(1, 1, P, t, \frac{t_{win}}{\Delta T}\right) \end{aligned}$$

- **Jumping window aggregation:** Here, one output tuple is produced during each re-evaluation period (independently of how many tuples lie within the time window t_{win}). As a number of $n = \left\lceil \frac{t_{win}}{P} \right\rceil$ tuples have been read during one re-evaluation period, the inverse of n is the average output per operator run.

$$\begin{aligned} \Phi &= (bi, bo, t, P, s) \\ &= \left(1, \frac{1}{n}, t, P, \frac{t_{win}}{\Delta T}\right) = \left(1, \frac{1}{\left\lceil \frac{t_{win}}{P} \right\rceil}, t, P, \frac{t_{win}}{\Delta T}\right) \end{aligned}$$

7.2.2. Sync-Join

The sync join operation $O_{sync-join}$ is a binary operation which takes two stream descriptors $\hat{S}_a$ and $\hat{S}_b$ as input. As a fundamental difference to existing database or data stream joins, it aims to produce meaningful output by joining only tuples which have the same production timestamp. First, different join strategies are compared for classification purposes.

Stream Joins in General

Table 7.2 describes existing joins based on the two conceptual steps *merge* and *probe*, and based on the semantics of their results. This can be seen as a small piece of related work.

Joins over database relations do not require an explicit merge step. The input relations are completely available at join execution time. All tuples of both relations have to be considered for probing. Thus, the probe step is complex because of the potentially high number of tuples.

Joins over data streams can be divided into *sliding window joins* and *sync joins*. Sliding window joins on data streams are well-known. They extend joins over database relations in two steps: first, they make them non-blocking and second, they limit the

	Database Join	**Stream Join**	
		sliding window join	**sync join**
merge step	**not required**; whole database relation available for joining; timestamp not an explicit attribute	**simple**; restrict the possibly infinite input streams by a window on each	**complex**; find the tuples that (temporally) fit together based on their timestamps (1:1 relationship); use interpolation and downsampling techniques
probe step	**complex**; probe all tuples of the first relation against all tuples of the second relation	**less complex**; probe all tuples of the first window against all tuples of the second window	**simple**; for the identified 1:1 tuple relationship make an additional test of the join predicate
quality metrics		**window size**; the larger the window on the input streams, the larger the result set and the higher the associated result quality	**inconsistency**; the more the tuple timestamps are 'tampered' with to associate the partner tuple, the lower the result quality (contrary to sliding window joins)

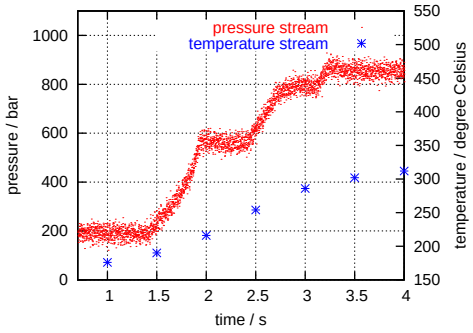
Table 7.2.: Join Classification

degree of the internal state the join operator has to maintain. Both objectives are achieved by applying a window (tuple-based or time-based) on the input streams and by performing the probe operation based on the data of the window. The larger the subset of the current stream, the more result tuples will be produced, which in turn is a measure for the result quality and optimization goal, respectively.

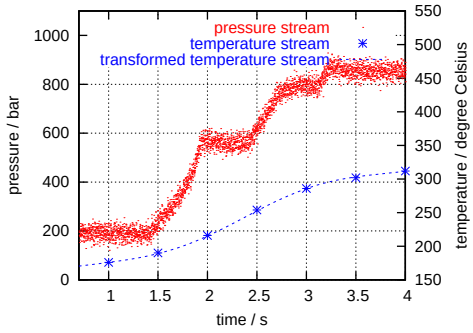
Sync Join Definition

The QStream model entirely focuses on the sync join approach. In this context, the concepts of [SFL05] are extended to select particular join strategies based on the stream types C_a and C_b of the stream descriptors $\hat{S}_a$ and $\hat{S}_b$ respectively.

In comparison to window joins, the sync join mechanism aims at producing consistent and semantically meaningful output–in particular by paying attention to the tuples' timestamps. The goal is to enable a time-consistent view on the modeled application scenario. This means that each output tuple consists of data which match the input tuples' timestamps as closely as possible. Consider the example stream data of Figure 7.8. There, two data streams of different sensors have to be joined. The temperature



(a) continuous streams with extremely different tuple timestamp distance (and data rate)



(b) create temperature stream tuples for each tuple contained in the pressure stream

Figure 7.8.: Sync Join Motivation

data stream contains only two tuples per second, whereas the pressure stream consists of about 1,000 tuples per second. The join's goal is to output triples containing a timestamp as well as an attribute instance value from both streams. If one does not want to lose data stream details, the result production should be oriented towards the stream with the higher tuple density (the pressure stream in the example). For that reason, the intermediate (missing) values of the temperature data stream are obtained using the resample operator. Then, the pressure and the temperature values are 'paired' and the result can be returned.

The general idea is to only include those attribute values in the result tuple which were produced at the same point in time. If one of the input streams cannot contribute data for that time point, two possibilities arise: Following *inner join semantics*, no result tuple is produced during the gap of one input stream. Instead, a punctuation message is created and written to the output stream.

Following the *(full) outer join semantics*, an output tuple will be produced but the missing attribute values are marked as $NULL$ values. If neither of the partial streams contributes any data, the output is terminated by a punctuation messages until new input data arrive.

The sync join operator $O_{sync-join} : (f_{\hat{S}}, f_{\hat{Q}_{content}}, \beta)$ has a *mode* and a *predicate* as input parameters: $\beta = (mode \in \{INNER, OUTER\}, predicate)$. The *mode* parameter controls the inner/outer join procedure and the *predicate* reflects a join predicate which the sync join applies in addition to the timestamp comparison. The join works symmetrically on the basis of the partial streams $S^P_{a,k} \in S_a$ $(1 \leq k < |S_a| - 1)$ and $S^P_{b,l} \in S_b$ $(1 \leq l < |S_b| - 1)$. In general, the number of partial streams contained in each of the input streams is greater than one. This extends the sync join to an n-nary join between a number of $n = |S_a| - 1 + |S_b| - 1$ partial streams with arbitrary stream classes.

An additional goal of the sync join is to keep the stream classes of each partial stream in the result. The only exception here is that–if partial event streams are involved in the sync join and the inner join mode is chosen–all other partial join streams (class CS or DS) change their classes to ES with tuples only defined at the timepoints of the tuples of the involved event partial stream.

Functional Description

The sync join procedure is divided into four consecutive steps, which are illustrated in Figure 7.9. After a short overview, each of the steps will be described individually.

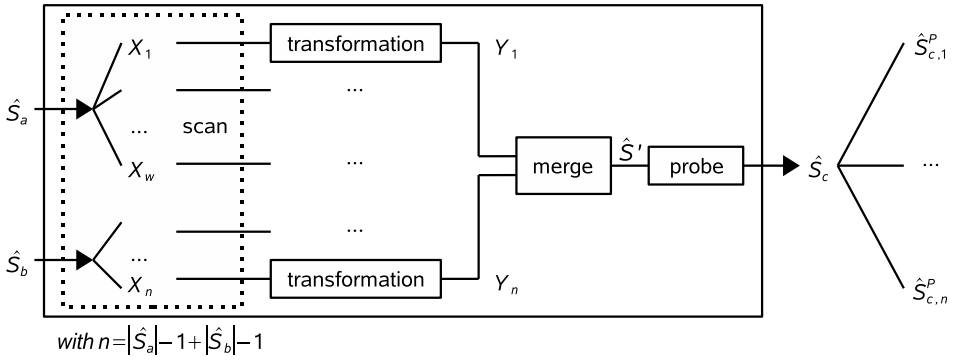


Figure 7.9.: Sync Join Processing Steps

1. **Scan** determines the timestamp candidates of the result stream by considering all partial input streams. The new minimal timestamp distance $\Delta T'$ is obtained based on the minimum timestamp distance of both input streams.

2. The **transformation** step pushes the partial stream either through the helper operator *resample* or through *reconstruct*–depending on whether the partial input streams are continuous or discontinuous. The goal here is to create all attribute values of all partial streams according to the result timestamp candidates acquired in the previous step. The exceptions are the event partial streams–they do not allow tuple reconstruction.

3. Thereafter, the transformed partial streams are **merged** into the result stream by paying attention to the inner/outer join mode, to the new minimum timestamp distance, and to the punctuation messages which may be included into the stream.

4. The **probe** step finally filters the join result following a user-given join predicate on another attribute than the timestamp. The probe step complies with the filter operator which is discussed at the end of this chapter.

The functional description of the sync join is given as

$$\begin{aligned} f_{\hat{S}}(\hat{S}_a, \hat{S}_b) &:= \hat{S}' \\ &:= f_{probe}(f_{merge}(f_{trans,1}((f_{scan}(\hat{S}_a, \hat{S}_b))_1, ..., (f_{trans,x+y}(\hat{S}_a, \hat{S}_b))_n))) \end{aligned}$$

The transformation functions of the individual steps are now described in detail. Note that the appropriate program components for scanning, transforming, merging and probing are pipeline-capable: they continuously read input data and continuously produce intermediate and final results respectively.

Scan Step

The goal of the scan step is to acquire timestamp information of all partial input streams for creating intermediate tuples during the following transformation step. The scan function

$$f_{scan} : (\hat{S}_a, \hat{S}_b) \mapsto (X_1, ..., X_w, ...X_n)$$

takes two stream descriptors $\hat{S}_a$ and $\hat{S}_b$ as input and creates a number of n intermediate results X_w. Such an intermediate result $X_w(S_w^P, e_w, c_w, \Delta T, \Delta T', SeqTS)$ consists of a partial stream S_w^P of one of the source streams, the attribute e_w and the stream class c_w (both of which belong to the corresponding source attribute), the minimal timestamp distance ΔT of the respective source stream, the timestamp distance $\Delta T'$ of the join result stream, and finally, a list of timestamp candidates $SeqTS$ of the join result.

The procedure of f_{scan} is the following: It iterates over all partial input streams S_w^P and considers each occurring timestamp as a result timestamp candidate by appending it to sequence $SeqTS$. The result stream's minimum timestamp distance $\Delta T'$ is simply determined as

$$f_{\hat{S},\Delta T}(\Delta T_{S_a}, \Delta T_{S_b}) = min(\Delta T_{S_a}, \Delta T_{S_b})$$

Note that $\Delta T'$ is determined only on the basis of the input meta data instead of the actual stream tuple timestamps. Tuple timestamps of the different partial streams may be closer together but the second-to-next merge step considers them to be equal within a time span of $\Delta T'$. Thus, the sync join result stream will definitely not contain tuples with timestamps closer than $\Delta T'$.

Transformation Step

During the transformation step, each partial stream S_w^P is transformed using the parameters $SeqTS$, ΔT and $\Delta T'$, which were provided by the previous scan step. When doing so, all tuples which may potentially be useful (which may potentially find a 'join partner' with regard to the join predicate) are created for the consecutive merge step.

The transformation function $f_{trans} : X_w \mapsto Y_w$ maps each intermediate input item X_w to output items Y_w $({S'}_w^P, e_w, c_w, \Delta T')$ by changing the partial stream's content in terms of creating intermediate tuples. The partial stream ${S'}_w^P$ follows the timestamp candidates $SeqTS$ and the new minimal timestamp distance $\Delta T'$. The input partial stream's schema e_w and stream class c_w are only passed by. Depending on c_w, the transformation step shows different behavior:

$$f_{trans}(S_w^P, e_w, c_w, \Delta T, \Delta T', SeqTS) :=$$

$$\left({S'}_w^P, e_w, c_w, \Delta T' \mid {S'}_w^P = \begin{cases} resample(S_w^P, \Delta T, \Delta T') & \text{if } S_w^P \models_C CS \\ reconstruct(S_w^P, SeqTS) & \text{if } S_w^P \models_C DS \\ — & \text{if } S_w^P \models_C ES \end{cases} \right)$$

If S_w^P is a continuous stream, the required intermediate tuples are created using the *resample* helper operation. This ensures that the partial stream's class stays the same when interpolating additional tuples. The resampling factor $\frac{p}{q}$ is determined using the old and the new minimum tuple distance: $\frac{p}{q}$ equals the pruned fraction of $\frac{\Delta T}{\Delta T'}$.

If S_w^P is discontinuous, additional DS tuples are created following the timestamp sequence $SeqTS$ by using the *reconstruct* helper operator.

For event streams ($S_w^P \models_C ES$) no tuples must be reconstructed because no attribute instance value is valid between two tuples of consecutive timestamps.

Merge Step

During the merge step, the partial streams are combined following either the inner or the outer join semantics. Thereafter, it constructs the result stream descriptor.

The merge function f_{merge} takes a number of n intermediate items Y_w containing elements of the form $({S'}_w^P, e_w, c_w, \Delta T')$ as input. It produces one result stream denoted by the stream descriptors $\hat{S}'(S', E', C', \Delta T')$, where S' stands for the merged result stream, E' and C' denote the result streams' schema and stream type, respectively, and $\Delta T'$ is for the new minimal tuple timestamp distance:

$$f_{merge}((Y_1, ..., Y_w, ..., Y_n), mode) := (S', E', C', \Delta T')$$

Result stream: The result stream $S'(T_1, ..., T_j, ..., T_n)$ contains tuples T_j where a tuple $T_j(a_{1,j}, ..., a_{n,j})$ may be constructed depending on the occurrence of a continuous partial stream within one of the source streams S_a and S_b respectively. The reason for the

dependency on continuous partial input streams is that–if a continuous partial stream exists–its timestamps must stay equidistant due to its stream class and thus–during the sync join process–other partial streams' tuples have to be adapted to the existing timestamps of the continuous partial stream. Therewith, QStream accomplishes its implicit goal of retaining the source stream characteristics as much as possible. The merge step may work

1. **asymmetrically, based on a continuous stream:** If at least one continuous partial stream S_w^P is involved, take the timestamp from each of its tuples as timestamp $a_{1,j}$ of the potential result stream tuple T_j.

   Then, 'wait' for tuples from the other partial streams for the time interval $[a_{1,j}, a_{1,j}+\Delta T')$.

   If a tuple from another partial stream $S_w^P (1 \leq w \leq n)$ arrives within that interval, set $a_{w,j}$ of the result tuple T_j to its attribute value.

   If no tuples from other partial streams S_w^P with a timestamp in $[a_{1,j}, a_{1,j} + \Delta T')$ exists and an outer join is to be applied ($mode = OUTER$), insert the value $NULL$ as $a_{w,j}$ of T_j. Otherwise, if an inner join is required ($mode = INNER$), throw away the prepared result tuple T_j and only output a punctuation message. Repeat the procedure with the next tuple from the selected partial continuous stream.

2. **symmetrically, if no continuous stream is involved:** If no continuous partial stream is involved in the join, take the timestamp of each arriving tuple as the timestamp $a_{1,j}$ of the potential result stream tuple T_j.

   Then, wait for tuples from the other partial streams for the time interval $[a_{1,j}, a_{1,j}+\Delta T')$. Note that not more than one tuple per partial input stream can occur within $\Delta T'$, as $\Delta T'$ was selected to be the minimum of both input streams. If a gap is contained in one of the input streams, it is recognized by the occurrence of punctuation messages.

   Proceed as in the former case, in dependence on whether or not a tuple from each partial stream has arrived (regarding the inner/outer join mode and regarding the handling of punctuation message).

   After the result tuple has either been written to the result stream or thrown away, take the tuple with the next timestamp (no matter from which partial stream of S') and repeat the procedure.

The result of the merge step is a pre-final stream descriptor $\hat{S}'$ which still has to be probed using the filter predicate (next step). The final schema E' as well as the final stream type C' are already assigned here:

Result schema: The resulting schema $E'(e_1, ..., e_w, ...e_n)$ is a sequence of the single attributes with e_w corresponding to the appropriate attribute of the partial stream from either S_a or S_b.

Result stream type: The result stream type C is determined on the basis of the former classes of the individual partial streams. If an outer join was applied, the stream classes do not change and $C = (c_1, ..., c_w, ..., c_n)$ with c_w corresponding to the appropriate stream class of the partial stream from either S_a or S_b. For an inner join, two cases must be distinguished:

- Only continuous and discontinuous partial streams have been involved: The stream classes of all partial streams remain continuous and discontinuous, respectively.
- Event partial streams are involved: As these partial streams could not be transformed to contain more tuples, the result stream's tuples are restricted to the event stream's tuples at maximum. Thus, the stream class of other participating partial streams–no matter if continuous or discontinuous–changes to ES as only the required amount of attribute values is 'picked out' of them for merging.

Probe Step

During the probe step, the function $f_{probe} : (\hat{S'}, predicate) \mapsto \hat{S}_c$ applies a user-given join predicate to the result stream descriptor:

$$f_{probe}(\hat{S'}, predicate) := \hat{S}_c(S'', E', C', \Delta T')$$

The stream S'' is built from S' by filtering out all tuples which do not follow the join predicate $predicate$. The resulting and final stream descriptor is denoted as $\hat{S}_c$

Non-Functional Operator Description

Based on the content-based quality description of the input stream descriptors $\hat{S}_a$ and $\hat{S}_b$, the content-based result quality descriptor $\hat{Q}_{content}(F, I)$ is determined.

Signal frequency F: The signal frequency is not reduced by the sync join operator. The result stream contains signal frequencies which are as high as the signal frequencies contained in its input streams. Nevertheless, the probe step at the end of the sync join procedure may have dropped individual tuples, but the probe step semantics comply with the filter operator and thus, the arising gaps do not reduce the signal frequencies by definition. For the result stream, the minimum of both input signal frequencies F_{S_a} and F_{S_b} is assigned as lower bound.

$$F' = \min(F_{S_a}, F_{S_b})$$

Inconsistency I: The inconsistency is increased only by the merge step. The amount of increase depends first, on the inconsistencies I_{S_a} and I_{S_b} of the source streams and second, on the duration which the merge step waits for tuples of the individual partial streams to arrive (this is the time $\Delta T'$). The partial streams' tuples of this time span are furtheron represented by only one new tuple with one timestamp assigned - this is

the reason for the inconsistency increase, which equals the new minimum tuple distance $\Delta T'$.

$$I' = I_{S_a} + I_{S_b} + \Delta T'$$

Figure 7.10 illustrates the upper bound of the inconsistency increase, simplified for the merge of two partial streams S_u^P and S_v^P originated from the source streams S_a and S_b, respectively. The tuples $T_u(t_u, a_u)$ and $T_v(t_v, a_v)$ of the partial streams S_u^P and S_v^P are selected to be joined. The tuples' timestamp difference is $\Delta T' = t_v - t_u$.

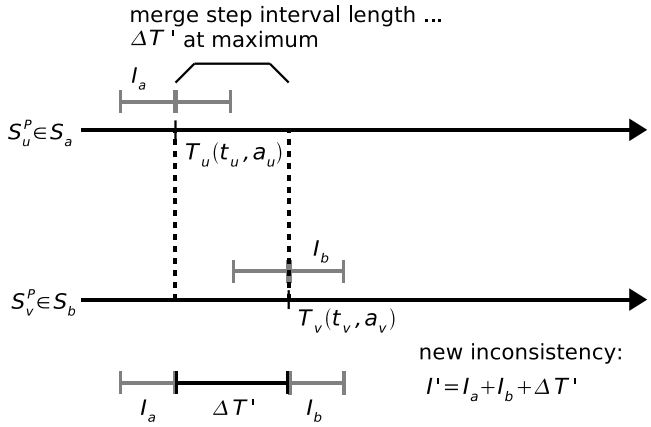


Figure 7.10.: Inconsistency Propagation for Sync Join

Operational Description

For the sync join, three different components are proposed to work in parallel. One for scanning and transforming each of the two input streams and one for the merge and the probe step together. The reason lies with the different processing speed requirements (in terms of periodicity) of the two input streams. The first and the second component (scanning and transforming) incorporate either the execution of the resample or the reconstruct operator–depending on the partial input streams' classes. The whole procedure works pipeline-based. The independently running components require individual operational descriptions. First, the operational description for components for scanning and transforming are given as

$$\begin{aligned} \Phi_{O_{resample}} &= (bi_a, bo_a, t_a, P_a, s_a) \\ &= (1, \frac{p}{q}, t, P, 2 \cdot L) \\ &= (1, \frac{\Delta T}{\Delta T'}, t_a, P_a, 2 \cdot L) \end{aligned}$$

and

$$\begin{aligned}\Phi_{O_{reconstruct}} &= (bi_b, bo_b, t_b, P_b, s_b) \\ &= (1, bo_{reconstruct}, t_b, P_b, 2)\end{aligned}$$

The number of average output tuples bo_b is based on statistics. It depends on the timestamp characteristics of the respective other stream. Due to the fact that $O_{reconstruct}$ may only insert tuples (no deletions), bo_b will be greater or equal to one.

Second, the component for merge and probe has to read two input streams and is described as

$$\begin{aligned}\Phi &= (bi_a, bi_b, bo_c, t_c, P_c, s_c) \\ &= (1, 1, bo_{merge-probe} t_c, P_c, 2)\end{aligned}$$

The merge-and-probe component reads the transformed input streams tuple by tuple and tests whether or not their tuple timestamps match (whether they are within a time span of $\frac{\Delta T'}{2}$ at maximum). Therefore, it only has to store the two input tuples of the respective input streams ($s_c = 2$).

The overall join selectivity (figuratively speaking) is influenced first, by the timestamp characteristics of the two input streams, second, by the input stream classes (CS, DS or ES), and third, by the selectivity of the join predicate.

7.2.3. Sampling

QStream proposes the sampling operator for probabilistically dropping stream tuples. It implements the *Stratified Random Sampling* technique ([Coc77]). There, a number of N input tuples (population) are read and n of them (randomly chosen) are passed by. The sampling operator $O_{sampling}(f_{\hat{S}}, f_{\hat{Q}_{content}}, \beta)$ requires the population size N and the number of tuples, n, to forward them as input parameters: $\beta = (n, N)$.

Functional Description

The sampling operator, changes the stream's content as well as the stream type using the transformation functions $f_{\hat{S},S}$ and $f_{\hat{S},C}$ respectively.

$$f_{\hat{S}}(\hat{S}) := (S', E, C', \Delta T \mid f_{\hat{S},S}(S) = S' \ , \ f_{\hat{S},C}(C) = C')$$

The stream schema E and the minimum timestamp distance ΔT are passed by without being changed.

Result stream: The stream content transformation function $f_{\hat{S},S} : S \mapsto S'$ decides which tuples T'_j of the input stream S will be written to the output stream S'. Formally, $f_{\hat{S},S}$ can be described as follows:

$$f_{\hat{S},S} \quad (\quad S((T_1,...,T_N),(T_{N+1},...,T_{2N}),...,(T_{(x-1)N+1},...,T_{xN})),n,N) := \\ ((T'_1,...,T'_n),(T'_{n+1},...,T'_{2n}),...,(T'_{x-1},...,T'_{xn})) \mid \\ T'_j \in \{T_{yN+1},...,T_{(y+1)N}\} \wedge y = \left\lfloor \frac{j}{n} \right\rfloor + 1$$

From the tuples of each population $(T_{(a-1)N+1},...,T_{aN})$ with $(1 \leq a \leq x)$, a sample $T'_{a-1},...,T'_{an}$ is taken. Thereby, each of the result tuples T'_j must be equal to one of the N tuples of the appropriate population $\{T_{yN+1},...,T_{(y+1)N}\}$ of the source stream.

Result stream type: The stream class of each partial input stream changes to an event stream ES because the temporal relationship of consecutive tuples gets lost if tuples are arbitrarily discarded:

$$f_{\hat{S},C}(C) := (ES,...,ES)$$

The sampling operator does not need to consider (forward or create) any punctuation messages, because it only outputs event streams. For ES, it is implicitly assumed that a 'gap' occurs after each tuple.

Non-Functional Description

The signal frequency cannot be propagated any longer, as it can only be assigned to continuous (partial) streams, and during sampling, the class of all partial input streams is changed to ES. In comparison, the inconsistency does not change as no tuples are merged and no timestamps are tampered with:

$$I' = I$$

Operational Description

The resample operator reads the input tuples in batches; each batch contains N tuples $(bi = N)$. Then, within each period, a number of n tuples is written to the result stream $(bo = n)$. In total, N tuples need to be stored in internal memory units:

$$\begin{aligned} \Phi &= (bi, bo, t, P, s) \\ &= (N, n, t, P, N) \end{aligned}$$

7.3. Stateless Operators

The operator repertoire of QStream is completed by the stateless operators *filter* (O_{filter}) and *projection* ($O_{projection}$). Both are described within the following section.

7.3.1. Filter

The filter operator $O_{filter}(f_{\hat{S}}, f_{\hat{Q}_{content}}, \beta)$ applies a user-given predicate to the attribute values of the stream tuples. The user-given parameter is the filter predicate: $\beta = (predicate)$.

Functional Description

The content transformation $f_{\hat{S},S}$ drops tuples which are not conform with the *predicate*:

$$f_{\hat{S}}(\hat{S}, predicate) \quad := \quad (S', E, C, \Delta T \mid f_{\hat{S},S}(S) = S')$$

The schema E and the minimum timestamp distance ΔT are not changed. Furthermore, the stream type C also remains unchanged by definition because–if the gaps do not occur very frequently–the stream class can still be assigned to the remaining stream fragments. Otherwise, if the original attribute value behavior is not clear any longer due to frequents gaps, the class of continuous or discontinuous partial streams could be changed to ES. The problem there is that one must know the stream behavior (or the gaps, respectively) over a longer time period to reason properly.

Result stream: The stream content transformation function $f_{\hat{S},S} : S \mapsto S'$ applies a filter predicate to all attribute values contained in the tuples of S. If the predicate evaluation delivers *true*, the tuple passes by. Otherwise, it is dropped. In case of the latter, a punctuation message T_P is inserted at the time the first tuple is dropped (Figure 7.11). The end of the gap is implicitly signaled by the occurrence of the next regular stream tuple.

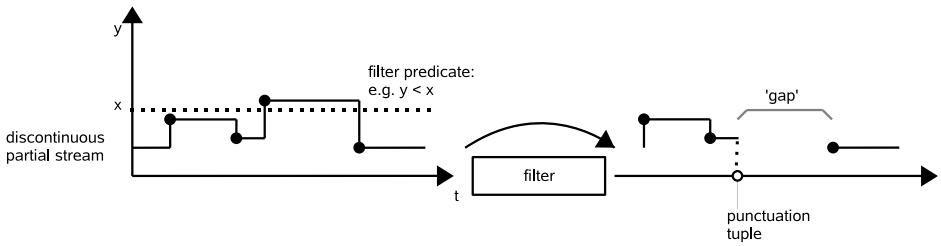


Figure 7.11.: Filter Operator Creating a Punctuation Message

A formal description of the content transformation is given as follows:

$$f_{\hat{S},S}(S(T_1, ..., T_j, ..., T_n), predicate) \quad := \quad \begin{cases} T_j & \text{if } eval(T_j, predicate) = true \\ T_P & \text{if } eval(T_j, predicate) = false \wedge \\ & eval(T_{j-1}, predicate) = true \wedge \\ & \exists i (1 \leq i \leq m) : S_i^P \models_C (CS \vee DS) \\ \emptyset & \text{otherwise} \end{cases}$$

Non-Functional Description

The signal frequency contained in continuous partial streams is not reduced, even though tuples which do not conform to the filter predicate have been dropped.

$$F' = F$$

The motivation is similar to the partial stream classes, which do not change by default: If the remaining stream fragments become too short, the partial stream's class may be explicitly changed to ES and thus, no signal frequency property can be assigned any longer.

Furthermore, no tuples are merged and no timestamps are tampered with. The stream's inconsistency remains unchanged, although the tuples which were responsible for the *maximum* inconsistency may have been dropped (the inconsistency property was not stored individually for each stream tuple and thus, it cannot be decreased if individual tuples are removed).

$$I' = I$$

Operational description: The filter operator reads the input stream tuple by tuple ($bi = 1$). The filter's selectivity determines how many tuples are written to the output stream. Thus, bo has to be obtained from statistics.

$$\begin{aligned} O_{OP} &:= (bi, bo, P, t, s) \\ &:= (1, bo_{filter}, P, t, s) \end{aligned}$$

7.3.2. Projection

The definition of the projection operator $O_{projection}(f_{\hat{S}}, f_{\hat{Q}_{content}}, \beta)$ reflects the semantics of the projection operator Π of the relational algebra. It transforms the tuples to a state conforming to the projection goal given as parameter $\beta = (attributes(e_1, ..., e_x))$. Thereby, partial streams whose attributes are not contained in the attribute list are deleted.

Functional Description

The content transformation function $f_{\hat{S},S}$ removes attributes which are not contained in the projection goal *attributes*. Thereby, the stream's schema E and the stream's type C are also changed by the transformation functions $f_{\hat{S},E}$ and $f_{\hat{S},C}$ respectively. The minimum timestamp distance ΔT remains unchanged:

$$f_{\hat{S}}(\hat{S}, SeqTS) := (S', E', C', \Delta T \mid f_{\hat{S},S}(S) = S' \ , \ f_{\hat{S},E}(E) = E' \ , \ f_{\hat{S},C}(C) = C')$$

Result stream: The stream content transformation function $f_{\hat{S},S} : S \mapsto S'$ passes by the timestamp attribute together with the attributes listed in *attributes*. A description is given as follows:

$$f_{\hat{S},S}(S, attributes) := [S_1^P, ..., S_i^P, ..., S_m^P \quad | \quad \forall i(1 \leq i \leq m) : \\ S_i^P \models_E E(e_1, e_2) \wedge e_2 \in attributes]$$

Punctuation messages do not need to be considered by the projection operator.

Result schema: The resulting stream schema equals the original schema, except for the attributes which are not in the *attributes* list.

$$f_{\hat{S},E}(E(e_1, ..., e_m), attributes(e_x, ..., e_y)) := (e_1, e_x, ..., e_y \ | e_x, ..., e_y \in attributes)$$

Result stream type: The resulting stream type is the sequence of stream classes of the remaining partial streams.

$$f_{\hat{S},C}(C(c_1, ..., c_m), attributes(e_x, ..., e_z, ..., e_y)) \quad := \quad (c_x, , ..., c_z, ..., c_y \ | \forall z(x \leq z \leq y) : \\ S_z^P \models_C c_z \wedge \\ S_z^P \models_E e_z \wedge \\ e_z \in attributes)$$

Non-Functional Description

The signal frequency of the remaining part of S is not reduced as only certain partial streams are removed.

$$F' = F$$

No tuples are merged and no timestamps are tampered with. The stream's inconsistency remains unchanged.

$$I' = I$$

Operational Description:

Similar to the filter operator, the projection operator reads the input stream tuple by tuple ($bi = 1$) and writes the projected tuples immediately to the output stream ($bo = 1$).

$$\begin{aligned} \Phi &:= (bi, bo, P, t, s) \\ &:= (1, 1, P, t, s) \end{aligned}$$

7.4. Summary

Finally, an example query is used to illustrate the standing query modeling aspects. A procedural query definition is given by Figure 7.12.

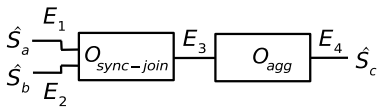


Figure 7.12.: Example Standing Query Q

The query Q contains a sync-join and an aggregation operator. The input streams are event streams and contain tuples of the form (ts_a, a) and (ts_b, b), respectively. The minimum timestamp distance is $\Delta T_a = 1s$ and $\Delta T_b = 2s$. The stream descriptors are

$$\hat{S}_a(S_a, (ts_a, a), (ES), 1s)$$
$$\hat{S}_b(S_b, (ts_b, b), (ES), 2s)$$

Furthermore, the content-based quality descriptors $\hat{Q}_{content,a}(F_a, I_a) = (-, 100ms)$ and $\hat{Q}_{content,b}(F_b, I_b) = (-, 75ms)$ belong to the stream descriptors $\hat{S}_a$ and $\hat{S}_b$, respectively. A signal frequency property F cannot be assigned to the input streams due to the event stream characteristics.

The dataflow through the operators is described by the edges of the DAG. They are annotated in Figure 7.12. There are

$$E_1 < -, O_{sync-join} >$$
$$E_2 < -, O_{sync-join} >$$
$$E_3 < O_{sync-join}, O_{agg} >$$
$$E_4 < O_{agg}, - >$$

Formally, the standing query Q is defined as

$$Q(\{O_{sync-join}, O_{agg}\}, \{E_1, E_2, E_3, E_4\})$$

Within this example, the individual operator descriptions are

$$O_{sync-join}(f_{\hat{S}}, f_{\hat{Q}_{content}}, \beta) \ , \ \beta = (mode) = (OUTER)$$
$$O_{agg}(f_{\hat{S}}, f_{\hat{Q}_{content}}, \beta) \ , \ \beta = (t_{win}, f, (f_{agg,1} f_{agg,2})) = (10s, 1, (AVG, AVG))$$

The sync join operator reads two input streams S_a and S_b and performs an outer join. Thereafter, the aggregation operator calculates the average values of a and b within a sliding window of $10s$. The result stream descriptor is denoted as $\hat{S}_c$ and is of the form

$$\hat{S}_c(S_c, (ts, a, b), (DS, DS), 10s)$$

The stream content has been changed to S_c by the query. The result schema is E_c and contains the timestamp as well as the attributes a and b. Both remaining partial streams are discontinuous due to the aggregation operation; thus, the stream type changes to (DS, DS).

The output of the sync join operator comes with a minimum timestamp distance of $\Delta T_{sync-join} = min(\Delta T_a, \Delta T_b) = 1s$. The consecutive aggregation operator re-evaluates the join window after every 10 seconds, which changes the timestamp distance to $\Delta T_{agg} = T_{re-eval} = 10s$. The resulting inconsistency of the sync join operator is calculated as

$$I_{sync-join} = I_a + I_b + \Delta T_{sync-join} = 100ms + 75ms + 1000ms = 1175ms$$

The aggregation operator increases the stream inconsistency further, which leads to a final inconsistency value of

$$I_{agg} = 2 \cdot I_{sync-join} + t_{win} = 2 \cdot 1175ms + 10000ms = 11175ms$$

Thus, the resulting content-based quality descriptor is $\hat{Q}_{content}(F, I) = (-, 11175ms)$, meaning that the attribute instance values a and b of the result stream do not necessarily match their result timestamp ts exactly; the appropriate attribute values may have been valid within the interval of $[ts - 11175ms, ts + 11175ms]$.

To summarize, all QStream operators are listed in Table 7.3 along with the user-given parameters and with the statistics they require. The QStream operators are QoS-aware and propagate the QoS metrics whenever possible. This allows for QoS negotiation by considering the resulting partial streams and the user-given QoS requirements. Furthermore, the operational description Φ provides the basis for resource calculation as it includes the internal operator resource requirements as well as the operator's input and output characteristics.

Operator	**Parameters**	**Statistics**	**Resulting Signal Frequency**	**Resulting Inconsistency**
helper				
$O_{resample}$	p, q		$min(F, \frac{p}{\Delta T \cdot q})$	$2 \cdot I + \frac{(q-1) \cdot \Delta T}{p}$
$O_{reconstruct}$	$SeqTS, \Delta T_{SeqTS}$	$bo_{reconstruct}$	—	I
stateful				
O_{agg}	$t_{win}, T_{re-eval}$, $(f_{agg,1}, ..., f_{agg,x})$		$min(F, \frac{1}{T_{re-eval}})$	$2 \cdot I + t_{win}$ (not for CS average aggregation)
$O_{sync-join}$	$mode, predicate$	$bo_{reconstruct}$, $bo_{merge-probe}$	$max(F_{S_a}, F_{S_b})$	$I_{S_a} + I_{S_b} + \Delta T'$
$O_{sampling}$	N, n		—	I
stateless				
O_{filter}	$predicate$	bo_{filter}	F	I
$O_{projection}$	$attributes(e_1, ..., e_x)$		F	I

Table 7.3.: Overview of QStream Operators

8. Integrated Cost Model and Scheduling Approaches of QStream

This section covers the QStream resource calculation approach for individual operators and queues as well as for whole standing queries. First, Section 8.1 introduces the extensions of the model of jitter-constrained periodic streams, which is denoted as JCP+. Thereafter, Section 8.2 presents different alternatives for scheduling a standing query instance and for trading off required resources for Quality-of-Service. The last part (Section 8.3) discusses the overall steps of the QStream resource calculation. It extends the generic resource calculation approach from Section 8.1 by including the alternative scheduling strategies from Section 8.2. Thus, final reasoning about resources required by a standing query instance is made possible.

8.1. The JCP+ Cost Model

The QStream resource calculation approach is based on the model of jitter-constrained periodic streams (JCP, [Ham97]), which attempts to obtain the inter-operator FIFO buffer as well as the delay time a producer operator needs to start ahead of its connected consumer operator; the eventual goal in this approach is to enable a continuous data flow. To extend this very generic approach, the specifics of the QStream application scenario are described in Section 8.1.1. Then, in Section 8.1.2 JCP is extended to *JCP+* by considering the calculation of specific resources and QoS metrics based on the data stream application requirements. A basic idea of that task was already given in [BSLH05]. In the last part, in Section 8.1.3, the cost approach is generally extended from a binary operator-operator relationship to a complete standing query instance.

The first purpose of JCP+ is to obtain resources which can be reserved at a resource manager of the operating system (OS) underneath. The resource calculation results must be compliant with the OS APIs, and the resulting QoS values must be suitable for negotiating user-specific requirements. For the sake of simplicity, FIFO buffer sizes as well as internal operator memory are always given in 'number of tuples.' In practice, these values can easily be converted into bytes considering each stream's schema information. Furthermore, delay and processing times are given in seconds and CPU utilization is described in percent.

The second and much more important purpose of JCP+ is to determine resource values that are sufficient to perform the planned query evaluation task: the lowest upper bound based on the input parameters is needed to cover the worst case and to provide QoS guarantees. Reserving resources without them being used would unnecessarily result in the rejection of standing queries or in the negotiation of a lower QoS than possible.

To summarize, the JCP+ model provides the worst-case resources in terms of *memory* and *processing time / CPU utilization* as well as the result QoS in terms of *output delay*.

8.1.1. Cost Model Assumptions

The calculation is divided into an *operator-based* and a *stream-based* part. The former is a rather trivial issue and consists of an operator's internal memory s and the processing time t, which an operator requires periodically; both are given by the operational parameter set Φ. Under the assumption of a *constant average data rate*, the operator-based calculation part simply consists of accumulating the costs required for the operators involved.

The stream-based calculation consists of determining intermediate buffer sizes as well as delays between the individual operators. Therefore, atomic buffer access of producer and consumer is assumed (no explicit buffer access time).

Extending the Runtime Description by Jitter Tolerance

To perform calculations for realistic application scenarios, QStream operators are allowed to *jitter* with regard to their data production process in the processing time t and in the amount of produced output data bo.

The *time jitter* is motivated by the fact that tuples will not be disseminated by a producer at equidistant points in time. Within the sensor streaming scenario, the sensor (as the first producer) may send the measured values a bit too early or too late (regarding the assumed constant data rate). Besides, the task of processing a fixed number of input tuples is not completely deterministic regarding time.

A *volume jitter* is caused by a varying number of produced tuples per run (bo is only an average value). Depending on the input data and on indeterministic operator functions, the operator may produce too many or too few tuples.

Generally, the amount of jitter (time as well as volume) is caused by changing environmental conditions in terms of input data streams. It is impossible to make a QoS-guarantee DSMS completely resistant to this influence as environmental changes cannot be foreseen in each case. Therefore, a pragmatic distinction between *micro jitter* and *macro jitter* is necessary. Micro jitter is included in the resource calculation and thus subject to JCP+. For that amount of jitter, the QStream DSMS can give guarantees regarding time-based QoS requirements. In comparison, resources for covering the macro jitter are not calculated by JCP+ and are thus not included in QStream's a-priori resource reservation.

Furthermore, the temporal scope of micro and macro jitter is different: Macro jitter rather refers to long-term changes of the input data (trends, periodic behavior). In contrast, daytime changes of acquired sensor data belong to macro jitter, whereas network delays or DSMS-specific processing time jitter are classified as short-term.

Considering micro jitter: We assume that the maximum deviation in time and in size of the micro jitter is known. Furthermore, the size jitter is a *cumulated* value. It means

that–if the operator has produced too much data during some of its runs–the operator has to produce fewer data during some of its other runs. Otherwise, the intermediate data rate would not be constant on average. In comparison, the maximum time jitter is maintained as a cumulated *and* as an absolute value. The concrete scheduling strategy determines which jitter characteristics to use.

In order to consider micro jitter by JCP+, the operational description

$$\Phi(bi_1[,bi_2],bo,P,t,s)$$

is extended to cover a time jitter regarding the average processing time t as well as a size jitter regarding the output batch size bo:

The triple $(\tau^{\perp},\tau^{\top},\tau^{\oplus})$ denotes the time taken by an operator when finishing a single production process too early $(\tau^{\perp})$ and too late $(\tau^{\top})$, respectively. $\tau^{\oplus}$ stands for the cumulative maximum processing time, which may be 'dammed up' at the operator. In the same manner, the parameters $(\sigma^{\ominus},\sigma^{\oplus})$ stand for the minimum and maximum cumulative volume jitter. The extended operational description is

$$\Phi^{*}(bi_1[,bi_2],bo,(\sigma^{\ominus},\sigma^{\oplus}),t,(\tau^{\perp},\tau^{\top},\tau^{\oplus}),P,s)$$

The operational parameters bi, bo and s are supposed to be known from the internally used algorithms. The other parameters–and sometimes also bo–depend on system characteristics (hardware capabilities) or on input data (data distribution). Therefore, an exhaustive set of statistics is supposed to be available which includes the processing time t and its time jitter $(\tau^{\perp},\tau^{\top},\tau^{\oplus})$ as well as the volume jitter $(\sigma^{\ominus},\sigma^{\oplus})$. Finally, the period length P is subject to the standing query instance evaluation speed, which is obtained in conjunction with the resource calculation and QoS negotiation.

Chapter 10 extensively explains the functioning of the QStream *statmon* component, which is responsible for gathering and maintaining operator instance and data stream statistics. Resource calculation for covering micro jitter within JCP+ is topic of Section 8.1.2. During this process, the amount of required resources generally increases with the size of the considered jitter.

Considering macro jitter: In comparison to micro jitter, macro jitter is intentionally not covered by the JCP+ calculations and thus not included in the worst-case resource reservation. Macro jitter is considered on a 'higher' level of standing query evaluation: If macro jitter in terms of longlasting changes occurs, the reserved resources for the standing query evaluation are not sufficient any longer and the DSMS signals exceptions in terms of buffer overflow or underrun. Thus, from time to time the query evaluation process must be adapted to new environmental situations. This includes new QoS negotiation as well as new resource reservation.

As a result, the principle of 'worst-case resource reservation' is weakened because it only holds for micro jitter. It follows that it is impossible to give any QoS guarantees regarding the macro jitter, because the macro jitter's reason lies in the unforeseeable input data stream. The issue of managing macro jitter through DSMS adaptation is topic of Chapter 9.

Standing Query Evaluation Speed

A standing query requires a fast, continuous and (with regard to QoS-guarantee systems) also a predictable evaluation. Moreover, a basic requirement of on-the-fly query evaluation is to keep up with the arriving data.

Connecting application time with real time: Within QStream, it is assumed that the time information within the tuples' timestamp attributes is directly 'connected' to the ongoing system time. In other words, this means that–if two source tuple timestamps differ by an amount of time x–these two tuples arrive at the DSMS with the same distance x (disregarding network delays) and thus must be processed in that manner. It is one of QStream's characteristics to work in *real application time*. Fast or slow motion as a kind of replay are not considered, only live data streams are.

Therefore, a standing query instance is scheduled for a fixed and pre-determined execution speed. The push-based processing paradigm of stream processing systems is resembled by pull-based (periodic) work of operator instances with all of them gracefully adjusted to the accurate speed.

The basis of the query evaluation speed is the data rate requirement R_{min} given by the user. It is part of the user's quality requirement $Req(F_{min}, I_{max}, R_{min}, D_{max})$. Here, R_{min} may either be given as an average or as a maximum value, depending on the data rate scheduling strategies, which will be discussed in Section 8.2.

Determining operator instance period length and intermediate data rates: The data rate requirement R_{min} equals the output data rate of the last operator instance of a query. The speed of all predecessor operator instances must be determined on that basis. The operator instance speed is described by its period length P. The smaller the period length, the more often the operator instance is executed and the faster the operator instance processes the input data stream. A lower bound for P is given by the time t, which is how long a run of the periodic operator work lasts. Obviously, t must be smaller than P.

Based on the required result data rate R_{min} of an operator instance and based on the amount of data bo which OI produces during each of its periods, the period length P can be calculated as $P = \frac{bo}{R_{min}}$. Then, the required input data rate R'_{min} of OI's predecessor can be determined by using the input batch size bi as $R'_{min} = \frac{bi}{P}$. Thereby, the data rate requirement R_{min} is *propagated upwards* the standing query instance from the last to the first operator(s) (through each operator path). This is illustrated in Figure 8.1. Assume a result data rate requirement of $100T/s$. The data rate reduction is given as $\frac{bo_3}{bi_3} = \frac{1}{2}$, $\frac{bo_2}{bi_2} = \frac{2}{5}$ and $\frac{bo_1}{bi_1} = \frac{1}{2}$. Thus, input data rates of $200T/s$, $500T/s$ and $1000T/s$ are required for operators O_3, O_2 and O_1 respectively.

With that concept, the required overall input data rate R^*_{min} for fulfilling R_{min} can easily be obtained individually for each operator path OP containing the operator instances $OI_1, ..., OI_n$:

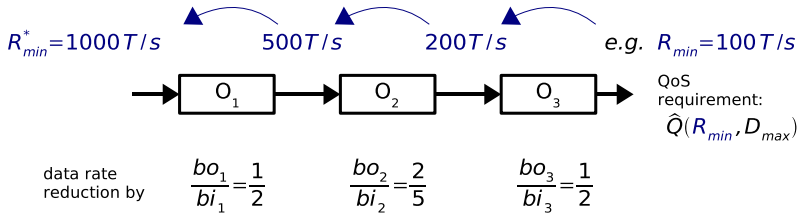


Figure 8.1.: QStream Data Rate Propagation Example

$$R^*_{min} = \frac{R_{min}}{\prod_{i=1}^{n} \frac{bo_i}{bi_i}}$$

Here, bi_i and bo_i belong to the respective operator instance OI_i and the ratio $\frac{bo_i}{bi_i}$ describes the *data rate reduction* of OI_i. In case of join operators, each path from the join input to source operators has to be considered separately.

Initial data rate adaptation: After R^*_{min} has been calculated, it must be adapted to the data rate provided by the data source. QStream favors sensor input data gathered by data acquisition hardware; thus depending on the peripheral devices, the adaptation may be done in either of the following two ways:

1. **Controllable data sources** can be parameterized and adjusted by the DSMS. Examples are Data Acquisition (DAQ) devices mounted locally. Such devices can be initialized and controlled by the DSMS - i.e. the source data rate requirements can be *pushed* up to the device and the QoS data rate requirements can be fulfilled as long as the device (hardware) is fast enough. Controllable data sources may be directly adapted to R^*_{min}.

2. A **non-controllable data source** sends its tuples independently from the requirements of the DSMS; the DSMS can only try to adapt to the source. Examples are distributed motes[1] or applications which just disseminate data in a broadcasting manner. Supposing that the data source works independently (delivering data at a rate of R_{source}), the DSMS (QStream) has to be adjusted initially to the data source properties.

In the case of non-controllable data sources, the adjustment has to be performed within the DSMS. Three possibilities arise:

1. If R_{source} equals R^*_{min}, obviously no adjustment is required.

2. If R_{source} is smaller than R^*_{min}, there is no chance to meet the data rate QoS requirements and QI must be rejected. Interpolating or reconstructing input tuples using the respective helper operators is not a good choice at this point: Even

[1] small independently working hardware devices equipped with sensors and processing units

though the input data rate could be increased, this does not hold for the information content and would thus not be beneficial in terms of QoS.

3. If R_{source} is larger than R^*_{min}, a *compensation operator* is inserted between the data source and the first operator of the appropriate path of QI.

The compensation operator's task is to reduce the data rate by the factor $\frac{R^*_{min}}{R_{source}}$. The query example of Figure 8.2 differs from the previous example (Figure 8.1) in that it has a higher data rate of the non-controllable source, which is $R_{source} = 2000T/s$. The data rate requirement of the first query operator remains constant at $R^*_{min} = 1000T/s$. Thus, a compensation operator for reducing the data rate by one half must be inserted between the data source and the first query operator.

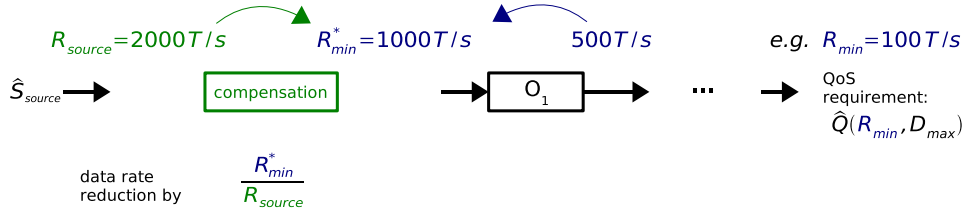


Figure 8.2.: Insertion of a Compensation Operator for Data Rate Adjustment

Depending on the stream class of the source stream descriptor $\hat{S}_{source}$, different techniques may be applied to accomplish the data rate reduction. For example, a sampling operator O_{sample} can be applied, Then, the sample operator's parameters $\beta = (n, N)$ are set equal to the pruned fraction of $\frac{R^*_{min}}{R_{source}}$.

QStream proposes to use the sampling operator only in case of event streams or if the input stream's attributes do not allow any aggregation. The reason is that the sampling operator would not maintain the data stream characteristics of continuous and discontinuous partial streams.

In case of continuous and discontinuous streams, the aggregation operator O_{agg} with the AVG aggregation function (jumping mode) should be used for data source adjustment instead of sampling. Due to the fact that QStream works in real application time, the window length t_{win} is set to the inverse of the required output data rate and the data reduction factor f must be equal to the quotient of the source data rate and the output data rate of the aggregation operator:

$$\begin{aligned} \beta &= (t_{win}, f, (f_{agg,1}, ..., f_{agg,x}) \\ &= \left(\frac{1}{R^*_{min}}, \frac{R_{source}}{R^*_{min}}, (AVG, ..., AVG)\right) \end{aligned}$$

For continuous partial streams, a resampling operation is implicitly applied as a substitute to AVG. The resampling helper operator parameters $\beta = (p, q)$ are equal to the pruned fraction of $\frac{R_{source}}{R^*_{min}}$.

The insertion of a compensation operator changes the structure of the standing query instance QI. Therefore, the standing query instance which results from the initial data rate adjustment is furtheron denoted as QI^*.

8.1.2. Generic JCP+ Calculation

The calculation of resources and QoS is based on the operator instances' behavior (periodicity, run times, jitter in time and in size) and results, first, in an amount of required intermediate buffer space B (resource value), and second, in the inter-operator start delay d (QoS value).

Data Exchange Concept

A key concept regarding QStream's data exchange is the *continuous data flow* through the standing query instance. It means that the data exchange must not be blocking. This is the basis of both, the original JCP cost model and QStream's JCP+.

Therefore, the main focus at this point lies on the combination of two consecutive operators (denoted as *producer* and *consumer*, or more formally, OI_i and OI_{i+1}; Figure 8.3). The producer periodically produces data and writes them to the buffer. The consumer reads the data periodically from the buffer. The producer is assumed to start its work earlier than the consumer (delay) to fill the buffer up to a certain level to ensure that the appropriate amount of data is available for consumption. The basic requirement for non-blocking behavior is that, on average, the output data rate of the producer equals the input data rate of the consumer.

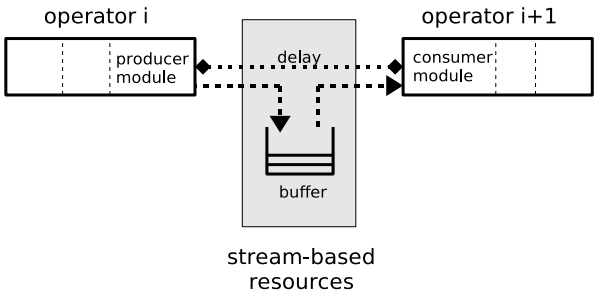


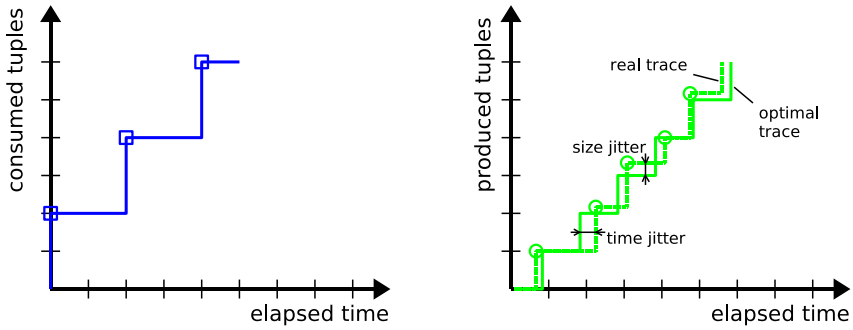
Figure 8.3.: Producer-Consumer Relationship Example

The intermediate buffer is required for two reasons: First, the read and write granularities of producer and consumer respectively have to be adjusted: For example, the producer may write single tuples to the buffer whereas the consumer may always try to read a batch of 10 tuples at once. In this case, the consumer would have to wait at least 10 times the producer's period length until it can successfully read the first data from the buffer.

The second reason for establishing a buffer is the considered micro jitter. There may be phases of high activity as well as phases of rather low activity regarding the data production process; both must be compensated by buffering data.

Describing the Operator Behavior by Traces

The behavior of a periodically running operator instance which consumes and produces at a constant average data rate can easily be visualized with *traces* drawn in a processing-time / data-volume diagram as illustrated in Figure 8.4(a) for a consumer operator. The time-axis shows the elapsed processing time, whereas the amount of produced or consumed tuples T is marked in direction of the y-axis.



(a) Operator instance (consumer) works periodically

(b) Operator instance (producer) works with jitter

Figure 8.4.: Operator Traces

The producer trace in Figure 8.4(b) extends the first diagram by annotating jitter of OI in time and size. The dotted line is the real trace and the solid line is the optimal trace, i.e. a trace without jitter in time or size. The difference between the two traces in direction of the time-axis is the time jitter. The difference in direction of the y-axis stands for the batch jitter.

If two operator instances (producer and consumer) are to be connected, the traces of both can be plotted in one single diagram of the same style (Figure 8.5). The dotted line is the trace of a producer, which produces on average one single tuple during each period. There is a producer jitter in time and size. The solid line is the jitter-free trace of a consumer which reads $bi_{i+1} = 2$ tuples from the buffer at constant time intervals. The traces are given for the case that both operator instances start working at time 0: the consumer's first action would be to read $bi_{i+1} = 2$ input tuples, whereas the producer would first process and then (after its processing time t) output $bo_i = 1$ tuple on average.

Obviously, the data exchange cannot work that way: To schedule the execution of both operator instances precisely, one has to ensure that every time the consumer reads a number of bi_{i+1} tuples, these tuples are already available in the buffer to avoid blocking.

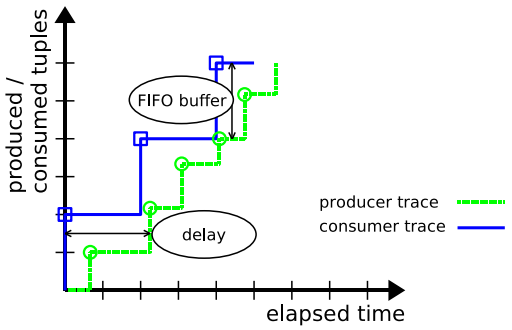


Figure 8.5.: Producer / Consumer Traces

Therefore, the consumer trace must be shifted to the right of the producer trace. This amount of time–the required consumer delay d–is represented by the maximum distance of the two graphs in direction of the time-axis. The maximum distance between the two graphs in direction of the y-axis represents the minimum required intermediate buffer size B. Both values are annotated in Figure 8.5.

Calculation Approach

In the first step, the delay and the buffer size are calculated without considering any jitter. To obtain both values in an easy fashion, one makes use of two linear approximation functions $u(t)$ and $l(t)$, which are shown as dotted lines in Figure 8.6.

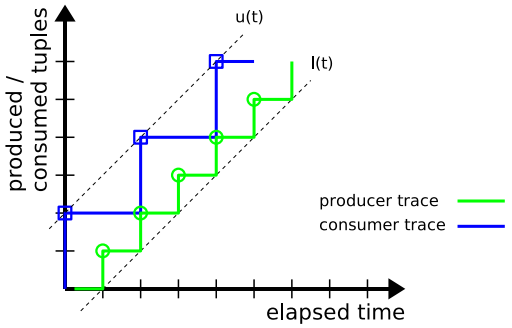


Figure 8.6.: Approximations

Function $u(t)$ describes the upper limit of the consumer trace, and function $l(t)$ stands for the lower limit of the producer trace. As lower bound, the delay d is determined as the maximum difference of these two linear functions in direction of the time-axis, whereas

the minimum buffer size B is set to the maximum difference of the two linear functions in direction of the y-axis.

The gradient of both linear functions $u(t)$ and $l(t)$ is the average data rate denoted as r. To obtain the complete description of these linear functions, an additional point for each function is required.

For the upper limit function $u(t)$, the point $P(t, y) = P(0, bi_{i+1})$ is known, because at time $t = 0$ the consumer starts to read a number of bi_{i+1} tuples, yielding the function $u(t) = r \cdot t + bi_{i+1}$. For the lower limit function $l(t)$, the point $P(t, y) = P(t_{i+1}, 0)$ is used, where t_i is the estimated run time of the producer obtained from OI_i's operational description. The lower limit function is $l(t) = r \cdot t - r \cdot t_i$. The delay may now be calculated using the horizontal distance of the two functions (given by the distance of the two functions' null values) with P_{i+1} being the period length of the consumer operator:

$$d = t_{0,l} - t_{0,u} = t_i + P_{i+1} \tag{8.1}$$

The distance $u(t) - l(t)$ of two traces in the direction of the y-axis yields a lower bound of the buffer size B, which can be calculated as follows (using the two functions' intersection points with the y-axis):

$$B = y_{0,u} - y_{0,l} = bi_{i+1} + r \cdot t_i \tag{8.2}$$

In order to consider time and size jitter, the calculations are extended. The time jitter is restricted to the absolute values $\tau^{\perp}$ and $\tau^{\top}$ because saved execution time cannot be effectively used in consecutive runs (due to the periodic operator work) and a working time longer than the period length P is not allowed for the time being. The cumulative time jitter $\tau^{\oplus}$ will be considered later together with the description of the particular scheduling strategies in Section 8.2.1. In contrast to the time jitter, the size jitters $\sigma^{\ominus}$ and $\sigma^{\oplus}$ are always used as cumulative values. In the operator trace in Figure 8.7, each vertex of the producer operator trace can be influenced by both kinds of jitter.

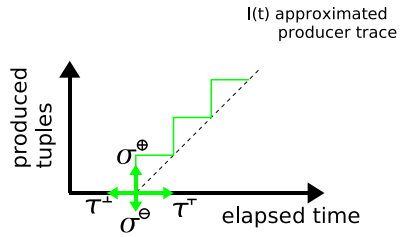


Figure 8.7.: Possible Occurences of Producer Jitter

Delay calculation: For delay calculation, only $\tau^{\top}$ and $\sigma^{\ominus}$ must be considered to avoid that the consumer tries to read data which has not yet been produced. The worst-case assumption here is that the maximum jitter of $\tau^{\top}$ (producer is too late) and $\sigma^{\ominus}$ (producer does not deliver enough data) occur at the same time, for example, at the beginning of the data exchange (Figure 8.8). There, the operator trace $l(t)$ passes into $l'(t)$.

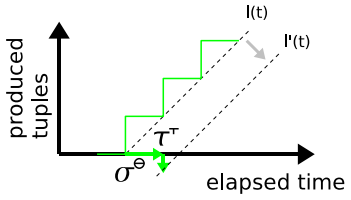


Figure 8.8.: Jitter Accumulation for Worst-Case Delay

The distances of the functions $u(t)$ and $l'(t)$ are again the basis for the calculation of d. For $l'(t)$, one uses the fixed point $P(t, y) = P(t_i + \tau^{\top}, -\sigma^{\ominus})$, thus

$$l'(t) : y = r \cdot t - (\sigma^{\ominus} + r \cdot (t_{O_{producer}} + \tau^{\top})$$

The consumer is not allowed to jitter in any way, so $u(t)$ remains unchanged, as

$$u(t) = r \cdot t + bi_{i+1}$$

Thus, by determining the distances of the two linear functions with the producer trace shifted to the maximum lower right position, one gets the following lower bound for delay d (as an extension of Formula 8.1).

$$d = t_i + \tau^{\top} + \frac{bi_{i+1} + \sigma^{\ominus}}{r} \tag{8.3}$$

Buffer size calculation: For buffer size calculations, $\tau^{\perp}$ and $\sigma^{\oplus}$ are important. Both values require an increase of the buffer size to avoid overflow. The worst-case assumption here is that the maximum jitters of τ (producer is too early) and $\sigma^{\oplus}$ (producer produces too much data) occur at the same time, for example at the beginning of the data exchange. Figure 8.9(a) depicts this situation.

In addition to the jitter values $\tau^{\perp}$ and $\sigma^{\oplus}$, the delay increment of the previous calculation step has to be factored in. A time jitter of $\tau^{\perp} + \tau^{\top}$ and a size jitter of $\sigma^{\ominus} + \sigma^{\oplus}$ must be considered as shown in Figure 8.9(b). The overall intermediate buffer size B is an extension of Formula 8.2 and can be calculated as

$$B = \left\lceil bi_{i+1} + \sigma^{\ominus} + \sigma^{\oplus} + r \cdot (t_i + \tau^{\perp} + \tau^{\top}) \right\rceil \tag{8.4}$$

The calculation of the final values of d and B has to be performed individually for each producer-consumer-relationship of the standing query instance in order to obtain the overall query resources. It is described in detail within the next section.

(a) Jitter accumulation for worst-case buffer enlargement

(b) overall buffer-influencing jitter

Figure 8.9.: QStream Jitter Accumulation

8.1.3. JCP+ Calculation for a Standing Query Instance

JCP+ attempts to be an *integrated* cost and QoS calculation approach including the overall resource requirements for a standing query instance QI^* as well as the query result QoS. Figure 8.10 is a generalization of Figure 8.3 and illustrates the necessary extension of the calculation approach (Formula 8.3 and 8.4) for obtaining the overall memory consumption M, the total delay D and the overall processing time requirements C caused by QI^*.

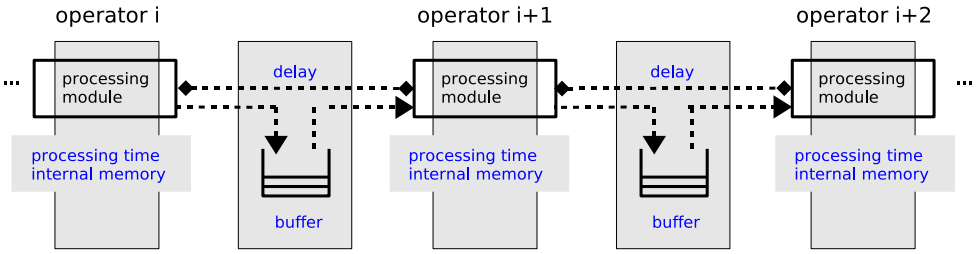


Figure 8.10.: QStream Overall Resources

The buffer B_i as well as the delay d_i are associated with the i-th producer operator OI_i of the query instnce ($1 \leq i \leq m$).

Delay

For obtaining the final delay D of a standing query instance QI^*, the DAG spanned by QI^* has to be traversed. For each operator path $OP_j \in QI (1 \leq j \leq u)$, the processing time t_i and the delay d_i of all n operator instances $OI_i \in OP_j$ ($1 \leq i \leq n$) has to be accumulated:

$$\forall OP_j \in QI^* : D_j = \sum_{i=1}^{n} (t_i + d_i) \tag{8.5}$$

The total delay is obtained by taking the maximum of all path delays D_j:

$$\begin{aligned} D_{QI^*} &= max_{j=1}^{u} D_j \\ &= max_{j=1}^{u} \sum_{i=1}^{n} (t_i + d_i) \end{aligned} \tag{8.6}$$

If different operator paths have different total delays (this is the general case), the sync join operators $O_{sync-join}$ at the branches of QI^* are unable to produce the desired result data. If the delay from the data source to the join operator of input tuples from the two join input streams differs, the corresponding tuples rarely have a chance of being joined as they arrive at the join in a time-shifted manner.

To overcome this problem, the tuples of the 'shorter' join input path can be delayed intentionally and directly before the join operator. The additional delay equals the delay difference of the two join input delays. As result, the total delay D_j of all operator paths OP_j is the same:

$$D_{QI^*} = D_1 = ... = D_u \tag{8.7}$$

In the example in Figure 8.11, three operator instances OI_1, OI_2 and OI_3 are shown. OI_3 is a sync join operator and OI_1 as well as OI_2 are connected to one of the join's input. The path delay after OI_1 is $D_1 = 100ms$ whereas the path delay after OI_2 is only $d_2 = 50ms$. An additional $50ms$ delay is added after OI_2 to ensure, that the respective tuples from both input streams arrive at the same time at the join ($D_1 = D_3 = 100ms$).

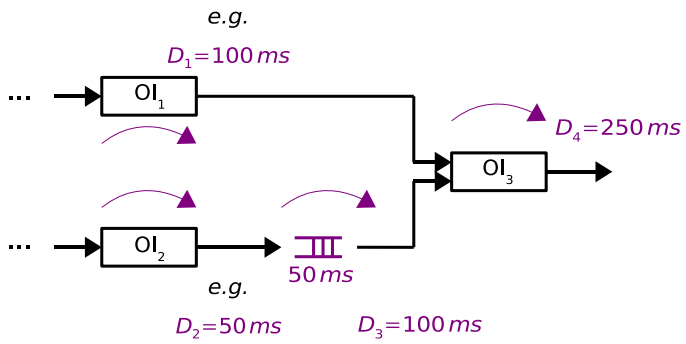


Figure 8.11.: QStream delay adaptation for a standing query instance QI^*

Applying this delay adjustment, the standing query instance QI^* does not need to be changed structurally as only an intermediate delay is increased. Also, this second adjustment does not conflict with the data rate adjustment of Section 8.1.1, as the extra delay does not influence the data rates and period length of other operators.

Memory

The overall memory consumption of the adjusted query instance QI^* consists of traversing QI^* and accumulating the internal memory s_i of all m operator instances $OI_i \in QI^*$ as well as the outgoing FIFO queue sizes B_i:

$$M_{QI^*} = \sum_{i=1}^{m} (s_i + B_i) \tag{8.8}$$

Processing Time

The overall processing time consumption C is also based on the traversation of the query instance QI^*. The individual processing times of the operator instances are not directly accumulated; only the relative utilization $\frac{t_i}{P_i}$ caused by each operator instance OI_i is considered. It follows that processing time requirements can only be given for a specific DSMS system environment as the individual processing times t_i are based on measurements of a specific system. The total average processing time requirement (in percent) of QI^* can be calculated as:

$$C_{QI^*} = \sum_{i=1}^{m} \frac{t_i}{P_i} \tag{8.9}$$

In this context, the operators' processing time jitters have not yet been considered. The calculation of the overall processing time will later be adapted to each of the specific scheduling strategies in Section 8.2.

To summarize, the resources have been considered in close relation to the Quality-of-Service requirements of a standing query instance. The overall resource requirements have to be tested against the system's available resources and the result QoS (delay) must be compared to the user-given QoS requirements.

Both, standing query instance resources as well as the associated QoS can be illustrated in a time/data volume-diagram. As can be seen in Figure 8.12, three example operator instances (sampling, filter and aggregation) are given along with their consumer and producer traces. The FIFO buffer as well as the intermediate delay are implicitly given by the distances between the operator traces of consecutive operators. Furthermore, the input data rate, the result data rate and the appropriate delays are annotated. To fulfill the QoS requirements, the 'final delay' must match the delay QoS requirement and the annotated 'result data' rate must match the standing query's data rate requirement.

8.2. Scheduling Strategies

Different possibilities exist for reserving the resources calculated with JCP+. The use of the processing time t as basis for JCP+ calculation, for example, is not the only available option; the sum of $t + \tau^{\top}$ as the worst-case execution time could be used as well. As

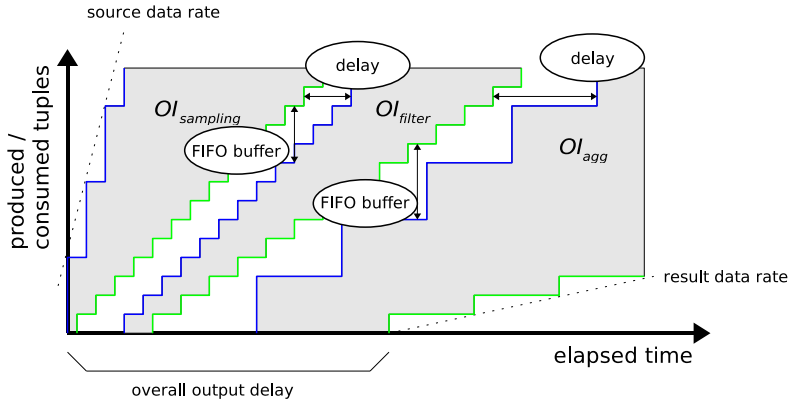


Figure 8.12.: Integrated Resource Management

another example, the maximum source data rate could be used for JCP+ instead of considering the average rate plus source jitter in time and volume.

Within the following subsections, a distinction is made between *run time scheduling* (Section 8.2.1) and *data rate scheduling* (Section 8.2.2). Both aim at mapping the JCP+ results to the operating system resource reservation API. In particular, the former handles different processing times as reservation basis, whereas the latter considers alternative data rates for planning. In addition, Section 8.2.3 announces a general optimization concept by considering *microperiods*.

The basis of scheduling the periodic operator work in QStream's scheduling strategies is Rate Monotonic Scheduling (RMS). It was described in the related work part of this thesis. With RMS, all operators receive fixed execution priorities inversely proportional to their respective period length. As a matter of fact, only one operator instance can occupy the CPU at a time. In this regard, operator instances with high priority may interrupt lower-prioritized ones but not vice versa.

8.2.1. Run Time Scheduling Strategies

The basis for scheduling an operator to work periodically is its period length and the working time within this period. The appropriate processing time of a run is called the *run time* t_{run} and lies in the range of

$$t - \tau^{\perp} \leq t_{run} \leq t + \tau^{\top}$$

The first question of scheduling processing times is concerned with the *deadline* not to be exceeded, and the second question focuses on the scenario that the deadline is exceeded anyway. Within the context of QStream, an operator's processing time deadline is associated with the beginning of its next period. Not every time an operator exceeds its average processing time, the deadline of this and of other operators is exceeded, too. First, only the respective operator and all lower-prioritized operators are influenced by an overextended working time. Second, it depends on the size of the jitter. If there is

enough processing time left until the next period start of an operator (if the CPU is not fully utilized), this time may be used for jitter compensation.

The goal of QStream's QoS-guarantee scheduling is to strictly avoid any miss of a deadline. If a deadline regarding the produced result tuples is exceeded anyway, the QoS cannot be guaranteed any longer and is considered broken. Then, it would be impossible to guarantee a certain system's reaction time or a minimum data rate of the output stream.

QStream proposes two run time scheduling strategies, both of which aim at meeting QoS guarantees in terms of maximum delays and minimum output data rates.

They are explained within the following two paragraphs.

Min Delay Strategy

Here, the jitter $\tau^{\top}$ is seen as an integral part of the processing time t_{run}, and thus, the time $t + \tau^{\top}$ has to be scheduled as run time during each period (Figure 8.13).

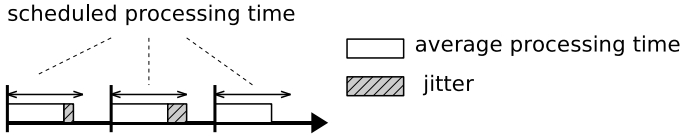


Figure 8.13.: Scheduled Processing Time in Min Delay Strategy

The jitter $\tau^{\top}$ may arise spontaneously and in full length within each of an operator's runs. This worst-case processing time reservation results in a higher calculated CPU utilization for a given data rate and lower data rates for given resources, respectively (because the scheduled processing times are given as worst-case processing times). The concrete formulas for obtaining C, D and M for the Min Delay runtime scheduling strategy are given later in Section 8.3.1. The advantage of the Min Delay strategy is that each operator instance finishes its run within the calculated worst-case time and no deadline is exceeded. This leads to a minimal inter-operator delay.

Max Throughput Strategy

In contrast to the previous approach, only the estimated operator run time $t_{run} = t$ within the operator's period is considered (Figure 8.14). Besides, the jitter component, which represents an exceeding of t, must now be cumulative constrained over all consecutive runs and is furtheron denoted as $\tau^{\oplus}$.

A benefit of the Max Throughput strategy is that either more operator instances (or standing query instances) are admitted or a faster network operation is achieved. The detailed resource calculation is given later in Section 8.3.1.

The drawback of considering only t is that–in situations of high load–operator instances will regularly and frequently exceed the given deadline in terms of the end of their current period. If such a deadline is exceeded at some point, there must be enough

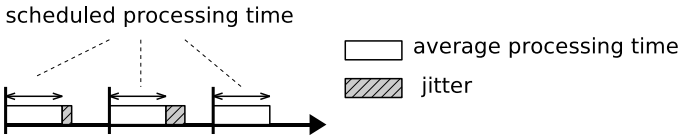


Figure 8.14.: Scheduled Processing Time in Max Throughput Strategy

free processing time within the following periods, so that this operator instance has the chance to get back to its 'normal operation'. An operator instance is in 'normal operation' if it works exactly in the prescribed period number at the current time, seen from an absolute starting point in time. If it is behind that expected period number, it is supposed to 'compensate its period' until it is back in 'normal operation.'

Although single operator deadlines may be exceeded, it is the absolute goal the keep the QoS guarantees. The consequences are:

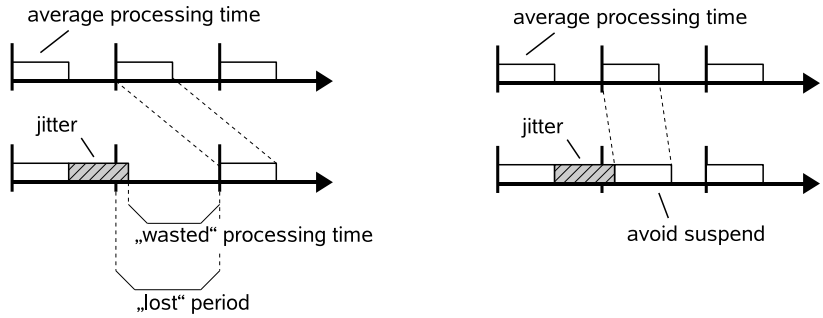
1. All DSMS operator instances which exceeded their deadline must be brought back to a normal operation state. There must be no durable effects of any deadline exceeding and of the subsequent compensation process.

2. It must be possible to calculate the upper (time) bound for the compensation process.

The remainder of this section aims to find solutions for both these requirements. In general, two possibilities arise if the time limit of the period is reached when executing the operator: the respective operator instance could be suspended if its period has elapsed (preemptive scheduling) or one could let the operator work until it has finished its run.

For data stream processing, it is not satisfying to interrupt an operator instance, even if too much processing time is consumed. The reason is that within a DSMS, there are dependencies between the operator instances reflecting the continuous data flow. If an operator instance cannot finish its work during its run and thus, is not able to output the result data, all other operator instances will be influenced by this situation: the ones which are positioned upstream to the suspended one may be blocked during their data exchange due to full intermediate buffers. Other operators located downstream to the suspended operator instance may be blocked due to empty buffers, i.e. due to missing data.

Instead of suspension, operators are given enough time to finish their work. This raises questions related to the time at which an operator will start its next run after it missed one or more deadlines. Figure 8.15 illustrates two possibilities. The operator could wait until its next regular period starts (Figure 8.15(a)). If so, it would lose a whole period every time a deadline was exceeded and thus it could not consume or produce enough data. The continuous data flow, and thus the QoS guarantees, would be broken. Furthermore, the time denoted as *"wasted" processing time* in Figure 8.15(a) would not be available for DSMS operators.



(a) start at next regular period (b) start immediately

Figure 8.15.: Period Exceeding

Therefore, an exceeded deadline is handled as shown in Figure 8.15(b). The operator is not suspended until it has caught up with all its missing period deadlines. Depending on the size of the jitter, an operator may not stop working for several consecutive periods. With Rate Monotonic Scheduling, all other operator instances with lower priority are suspended from work during that time. This requires specific buffer and delay extensions, which are described later in Section 8.3.

As a consequence of RMS and deadline exceedings, other lower prioritized operators may miss their deadlines, too. They do not necessarily work too long, but they are unable to start their next run at the scheduled point in time and therefore, they might not be able to finish before their period is finished either. It has to be clarified where each of the disturbed operator instances gets the time for compensating its period from. First, following QStream's assumptions, the jitter $\tau^{\oplus}$ is cumulatively constrained. This means, that if an operator instance has worked too long once, the same operator instance has to prematurely end one or several of the following run(s) for compensation. Second, if the CPU resources are not utilized 100%, there will be some processing time left during each period, which all disturbed operator instances may leverage for compensating their periods. This is a realistic assumption because due to running non-real-time processes, the CPU resources *must* never be completely occupied by running operator instances.

Jitter Compensation with the Maximal Throughput Strategy

One of the goals of the Max Throughput strategy is to give an upper bound of how much time it takes until all operator instances have finished their compensation process after one operator instance received a certain jitter and thus exceeded its deadline. This amount of time must always be compensated between each pair of consecutive operators; this means that even if a single operator instance consumes too much working time within its period, the arising jitter must not effect any of the following operators. Again, this is a criterion of the continuous data flow and for a QoS-guarantee DSMS.

It is impossible within the JCP+ resource model to describe the compensation process on the basis of the cumulative constrained jitter, because the jitter characteristic is not

known; only its (cumulated) maximum is avilable. Thus, one relies on the remaining CPU time for compensating a certain jitter and gives an upper bound of how long the compensation process takes. In a first step, the focus lies on a single operator instance OI with an average processing time t and a period length P. The task is to determine the time t_{comp} after which OI has compensated a jitter of length $\tau^{\oplus}$ (the maximum is assumed; thus, $\tau^{\oplus}$ may occur in full size during one run). A second step generalizes to an arbitrary number of concurrently running operator instances.

One-Operator approach: If a time jitter $\tau^{\oplus}$ outlasts one or more of the following operator instances' periods, OI will not be able to perform its regular work during that time. Figure 8.16 depicts that situation. The operator instance receives a jitter of length $\tau^{\oplus}$ which outlasts the first and the second period. Now, the time denoted as *"lost" time* must be compensated to bring the operator back to its normal operation state.

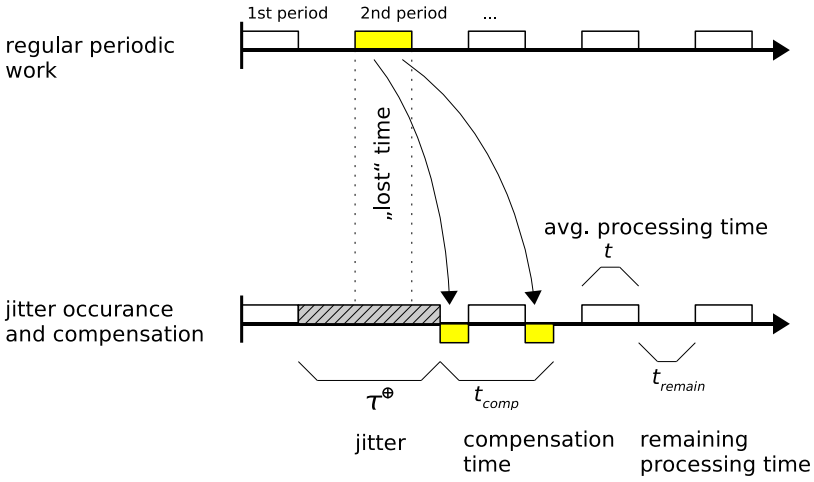


Figure 8.16.: Compensation Time Using Max Throughput Strategy

Therefore, first, the CPU time t_{remain}, which remains idle during each of OI's periods P (the time the operator instance may use for the compensation process), is expressed as the difference of the operator instance's period length P and the average run time t:

$$t_{remain} = P - t \tag{8.10}$$

Second, the compensation time is determined. It consists of n times the operator instance's period length P (if the jitter τ cannot be compensated until the end of the current period) plus a remainder. In the example in Figure 8.16, the compensation process is finished after one period ($n = 1$) plus three quarters of the second period (due to the operator instances' regular work within the second period). Formally, t_{comp} is calculated as stated in Formula 8.12. The length of the compensation time t_{comp} is based on two summands, with the first summand denoting the whole-numbered operator

instance period lengths P which the jitter compensation time outlasts. The second summand represents the remainder regarding the jitter compensation time and the first summand.

$$\begin{aligned} t_{comp} &= n \cdot P + (\tau^{\oplus} - n \cdot t_{remain}) \\ &= \left\lfloor \frac{\tau^{\oplus}}{t_{remain}} \right\rfloor \cdot P + \left(\tau^{\oplus} - \left\lfloor \frac{\tau^{\oplus}}{t_{remain}} \right\rfloor \cdot t_{remain} \right) \end{aligned} \tag{8.11}$$

Multiple-operator approach: The calculation is now extended to an arbitrary number of operator instances $OI_2, ..., OI_p$, which are influenced by OI_1's deadline exceeding. It is assumed that the number of an operator instance is in reverse order to its priority; thus, OI_1 has highest priority and OI_p the lowest one. There, p refers to the operator priority instead of the operator position.

Figure 8.17 illustrates the orthogonality between operator position and order of priority: A tuple (x_i, p_i) can be annotated at each operator instance. x_i defines the position of the operator instance regarding the data flow, whereas p_i stands for the execution order of the operator instance runs within a global repetitive cycle. The operator position x_i is independent of the operator priority p_i. If an example standing query instance is given as $QI(\{OI_1, OI_2, OI_3, OI_4\}, \{(-, OI_1), (OI_1, OI_2), (OI_2, OI_3), (OI_3, OI_4), (OI_4, -)\})$, the dataflow order is given as $x_1 < x_2 < x_3 < x4$, whereas the priority order could be $p_3 < p_4 < p_1 < p_2$, for example.

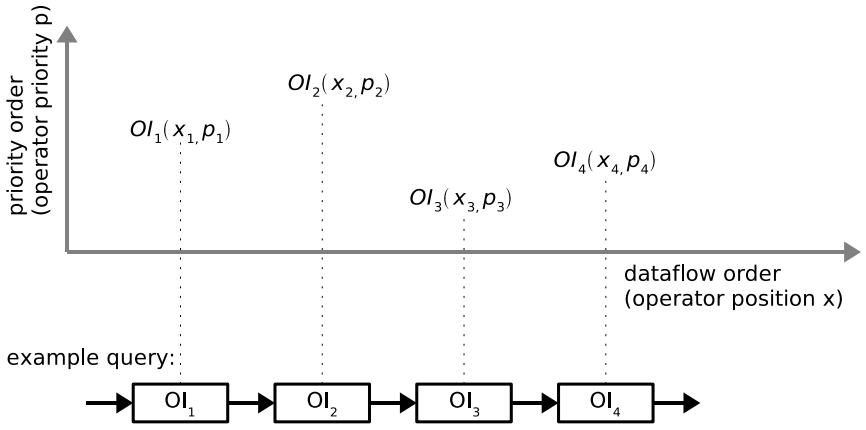


Figure 8.17.: Orthogonality of Dataflow Order and Priority Order

Furthermore, a distinction is made between an operator instance's jitter $\tau_p^{\oplus}$ (part of the operational description) and the overall jitter τ_p^{all}. The overall jitter τ_p^{all} of operator instance OI_p is caused by the jitter of other operators which are higher prioritized than the current one. These higher-prioritized operators exert influence due to their priority and due to the applied scheduling algorithm. The overall jitter τ_p^{all} may be recursively

expressed as the sum of the jitter $\tau_p^{\oplus}$ of operator instance OI_p, the overall jitter of operator instance OI_{p-1}, and the compensation time which OI_{p-1} needs:

$$\tau_p^{all} = \tau_p^{\oplus} + \tau_{p-1}^{all} + t_{comp,p-1}$$

Figuratively speaking, operator instance OI_p may start its next regular run (and its own compensation) when the delay *and* the compensation of its predecessor (in priority) have finished. Now, the compensation time $t_{comp,p}$ of the p-th operator is derived. The remaining CPU time is $P_p - t_p$ if no other operator instances with higher priority run concurrently. Otherwise, the processing times of these higher-prioritized operators have to be considered, too. This implies that the remaining CPU time $t_{remain,p}$ has to be determined depending on all higher-prioritized operators by simply subtracting their accumulated processing time (per period) from the available time (period length). Algorithm 1 describes this procedure.

Algorithm 1 Determining free processing time

Require: current operator O_p,
higher-priorized operators O_1, ..., O_{p-1},
initialization $t_{remain,p} = P_p - t_p$

```
2: for i := 1; i ≤ p − 1; i = i + 1 do
     x := P_p DIV P_i
4:   y := P_p MOD P_i
     t_remain,p := t_remain,p − x · t_i
6:   if y ≥ t_i then
       t_remain,p := t_remain,p − t_i
8:   else
       t_remain,p := t_remain,p − y
10:  end if
   end for
```

Additionally, Figure 8.18 illustrates how processing times of higher-prioritized operator instances decrease the free CPU time which OI_p could have used for its compensation process. The operator instances' period lengths and run times are annotated as P and t respectively.

Initially, the remaining time of operator instance OI_p is $P_p - t_p$. The higher-prioritized operator instances (OI_{p-1} and OI_1 in this case) decrease the remaining time as these operator instances continue working unpersuaded.

In analogy to the situation of an independently running operator instance, the time for compensating the jitter is expressed similar to Formula 8.12 with the exception that the overall jitter τ_p^all is used instead of $\tau^{\oplus}$.

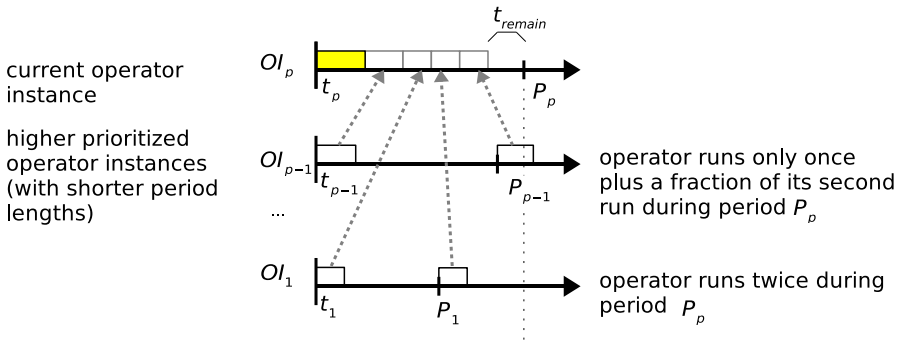


Figure 8.18.: Determining Free Processing Time

$$\begin{aligned} t_{comp,p} &= n \cdot P_p + (\tau_p^{all} - n \cdot t_{remain,p}) \\ &= \left\lfloor \frac{\tau_p^{all}}{t_{remain,p}} \right\rfloor \cdot P_p + \left(\tau_p^{all} - \left\lfloor \frac{\tau_p^{all}}{t_{remain,p}} \right\rfloor \cdot t_{remain,p} \right) \end{aligned} \tag{8.12}$$

The formulas for calculating the intermediate delay d and buffer size B as well as the total delay, the overall memory consumption M, and the CPU utilization C of a standing query instance QI^* running at Max Throughput strategy are more complex than with Min Delay strategy. Therefore, they are presented later in Section 8.3.1, where the resource calculation depending on both the selected run time and the data rate scheduling strategy is given.

Example of a Compensation Process

To get a better impression of how a compensation process in MT strategy works, an example is given (Figure 8.19): Three operator instances (*source*, OI_{filter} and *sink*)

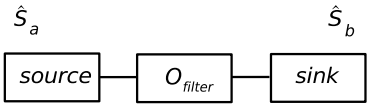
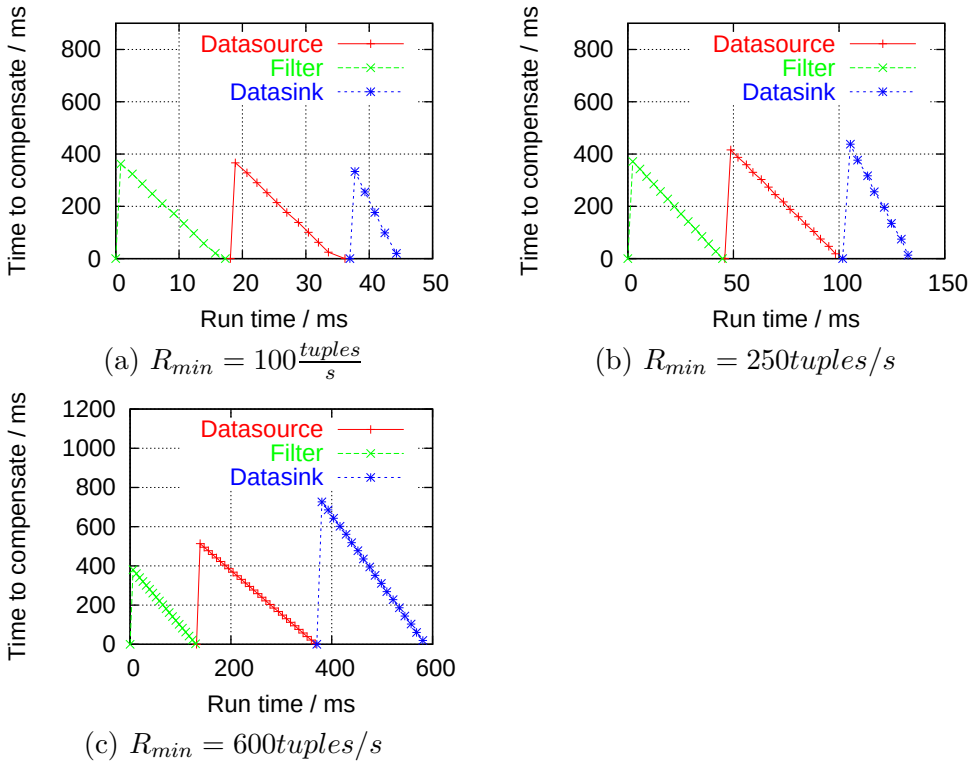


Figure 8.19.: Example Standing Query Instance

work concurrently. The filter operator has the highest priority, i.e. the data flow order ($source \rightarrow OI_{filter} \rightarrow sink$) is different from the order of priorities ($OI_{filter} \rightarrow source \rightarrow sink$). If the standing query instance receives an extraordinarily high jitter, the filter will start the compensation process first, followed by the data source and the data sink. The compensation process itself is illustrated within the diagrams of Figure 8.20 with a query running at different data rates ($R_{min} = 100, 250, 600 tuples/second$). The x-axis denotes the runtime and the y-axis denotes the time an operator instance still has at its disposal for the compensation to get back to normal operation.



(a) $R_{min} = 100\frac{tuples}{s}$

(b) $R_{min} = 250 tuples/s$

(c) $R_{min} = 600 tuples/s$

Figure 8.20.: Example Compensation Process of MT Strategy (from [SLL05])

Each of the three compensation processes is initiated by a filter jitter $\tau_{all} = 400ms$. Thereafter, the filter starts the compensation process, which last from $20ms$ (Figure 8.20(a)) up to $130ms$ (Figure 8.20(c)), depending on the remaining CPU time (and thus on the current data rate). The compensation processes of the other two operator instances start immediately after the filter jitter has been compensated. The length of the compensation time (t_{comp}) depends first, on the amount of τ_{all}, second, on the remaining CPU time and third on the time which an operator instance has lost during the jitter τ_{all}. Therefore, in all example configurations, the compensation time of the data source takes longer than the compensation time of the data sink, even though the data source received the smallest jitter.

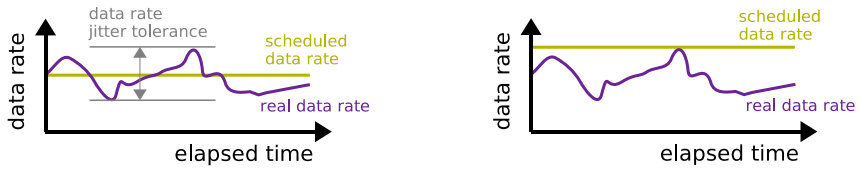
Comparison of Run Time Scheduling Strategies

A standing query instance running in Max Throughput strategy may achieve a high data throughput due to the fact that only the average processing times t have influence on the calculation of the CPU utilization. Compared to the Min Delay strategy, this comes at the cost of the output delay, which has to be calculated as the sum of the operators' processing times and the enlarged delays in between the operator instances (which have to be planned to compensate an overall jitter τ^{all}).

8.2.2. Data Rate Scheduling Strategy

To use JCP+ in a flexible way in various application scenarios, the data rate basis for planning and for calculation is determined by the *data rate scheduling strategy*. Up to now, the standing query evaluation speed and the initial data rate adjustment have been based on average rates R_{min} and R_{source}, respectively. The average incoming as well as all intermediate data rates had to be constant but the considered jitter in processing time and in the amount of output data implicitly caused a data stream jitter, too. The larger the jitter, the more resources were additionally required to enable continuous data flow.

Thus, to efficiently handle even large jitter and to support more sporadic data exchange behavior (like event signaling by a data stream), QStream proposes two different strategies for handling data rates, which are compared in Figure 8.21.



(a) average data rate scheduling (b) maximum data rate scheduling

Figure 8.21.: Data Rate Scheduling Strategies

Avg Data Rate Strategy

The ADR strategy schedules for continuous data flow. The jitter is supposed to be small in comparison to the average data rate. The speed of the operator instances and the intermediate queues are set up in a way that the operator instances are not blocked due to a full output buffer or due to an empty input buffer at any time.

The ADR strategy can be combined with both of the run time scheduling strategies (MD and MT). Details on how to obtain the specific resources in either case are presented in Section 8.3.1.

Max Data Rate Strategy

If the data is assumed to flow through the DSMS sporadically, the MDR strategy is more appropriate. The DSMS schedules and reserves resources for the maximum occurring data rate. Thereby, the query instance evaluation speed is generally higher than with ADR strategy, as it is based on a higher data rate assumption. This results in shorter period lengths and thus in higher CPU utilization.

Due to the scheduled maximum, most of the time, the input queues of operators are empty (and thus cause blocking). In contrast, the output queues must always allow for the operator to write data without blocking. The calculation formulas of C, D and M are given later in Section 8.3.1.

Comparison of Data Rate Scheduling Strategies

On the one hand, with the Average Data Rate strategy, the operators may run slower than with the Max Data Rate strategy, which causes lower CPU utilization. On the other hand, intermediate FIFO queues have to be allocated more space with the Average Data Rate Strategy. Thus, the user can select an appropriate data rate strategy as well as a run time strategy depending on the application requirements and available resources.

8.2.3. Scheduling Optimization: Concept of Microperiods

During their runs, the operators perform some work like filtering, manipulating or joining tuples. Focused on a single tuple, the scheduling overhead exceeds the time required to perform the operator instance runs. The concept of *microperiods* avoids scheduling overhead by considering a whole group of operator instance runs for calculating resources and for scheduling. Microperiods are the runs which an operator instance performs during one larger period of time. Only the larger period is scheduled by applying QStream's scheduling concept. If the user is not interested in QoS guarantees for each individual output tuple, QoS requirements may be specified for a whole group of tuples instead by choosing an appropriate number of microperiods.

When considering a group of operator instance runs, the processing time t refers to the processing of the whole group, and the length of an operator instance's period may be determined on that basis. In detail, this means that an operator reads, processes, and writes a whole group of tuples during one run. The operator description's values for input and output batch sizes, *bi* and *bo*, may only be seen as one factor for the number of group elements which are to be read or written. The other factor is the number of microperiods MP which are performed within one (larger) period.

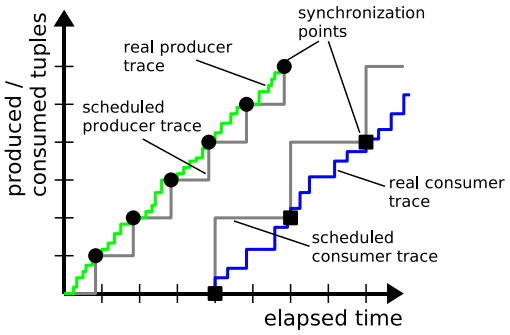


Figure 8.22.: Operator Trace with Microperiods

Figure 8.22 illustrates the traces of a producer and its connected consumer. The x-axis shows the elapsed time, and the y-axis represents the amount of transferred tuples. For each of the two operator instances, the real traces of producing or consuming, respectively, as well as the scheduled traces are shown. The *synchronization points* define the granularity of scheduling an operator instance's work. A step in direction of the

x-axis symbolizes the time of a period, and a vertical step stands for reading or writing a number of tuples.

Regarding the QoS negotiation process, the user may specify the maximum number of microperiods, MP_{max}, as a measure for the amount of data which other (time-dependent) QoS requirements refer to. The quality request which was introduced in Section 6.2.4 is extended to

$$Req(F_{min}, I_{max}, R_{min}, D_{max}, MP_{max})$$

Figure 8.23 sketches the relationship between the number of microperiods, MP, the required resources for query evaluation and the Quality-of-Service (regarding the microperiods): The larger MP, the smaller the QoS becomes, as time-based QoS guarantees refer to the large MP and fine-grained QoS negotiation is not possible any longer. The benefit of a larger value for MP is the lower resource requirements due to reduced scheduling overhead.

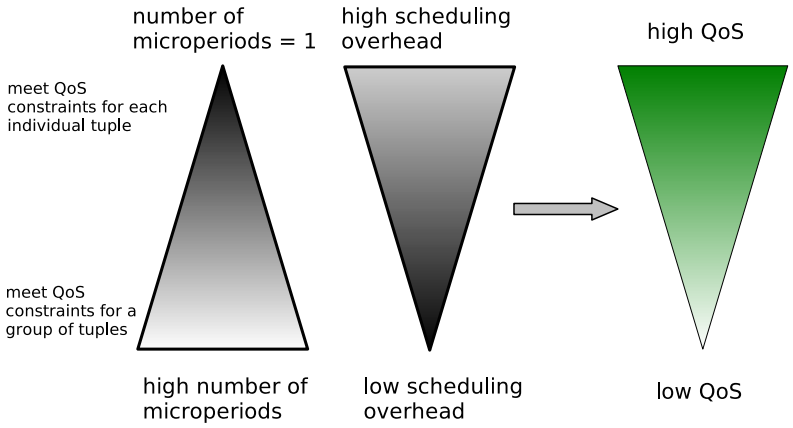


Figure 8.23.: Influence of Microperiods on QoS

The JCP+ resource calculation can be easily adapted to a number of microperiods $MP > 1$ by considering the whole group's processing time as t and the input and output batch sizes (bi and bo) multiplied with MP as the amount of consumed and produced data, respectively.

8.3. JCP+ Adaptation

This section combines the JCP+ calculation of Section 8.1 with the specific scheduling strategies of Section 8.2. First, within Section 8.3.1, the resource calculation (CPU utilization and overall memory consumption) for the different combinations of scheduling strategies is performed. Then, Section 8.3.2 extends Section 8.3.1 by focusing on the different states in which an operator can be if Max Throughput run time scheduling is used. Finally, Section 8.3.3 acts as a summary of Chapter 8 and explains the procedure

of resource and QoS calculation in conjunction with testing against available resources and the required QoS.

8.3.1. Scheduling-Strategy-Specific Resource and QoS Calculation

The application and combination of different scheduling strategies influences the JCP+ calculation and thus the resulting amount of resources and the achieved Quality-of-Service. Table 8.1 states the possible combinations:

	Run Time Scheduling	
	Min Delay	Max Throughput
Data Rate Scheduling		
Avg Data Rate	configuration I	configuration III
Max Data Rate	configuration II	configuration IV

Table 8.1.: Combination of Scheduling Strategies

All of the four combinations may be used. To clearly distinguish the Average Data Rate strategy from the Max Data Rate strategy (which differ in the period length basis), we denote the (large) period length of the former as $P_{i,ADR}$ and the (small) period length of the latter as $P_{i,MDR}$.

Furthermore, the identifier m always refers to the total number of operator instances of a query instance QI^*, whereas the identifier n denotes the total number of operator instances within an operator path $OP \in QI^*$.

Configuration I: Min Delay Runtime with Avg Data Rate Scheduling

If the Average Data Rate strategy is combined with the Min Delay strategy, Formulas 8.6 (maximum path delay) and 8.8 (overall memory), as proposed by the generic calculation, are directly used for obtaining the total delay D and the overall memory size M. The CPU utilization calculation is based on Formula 8.9. The processing times t_i as well as the maximum jitter values $\tau_i^\top$ are accumulated in relation to the respective operator instance period length $P_{i,ADR}$

$$C_{QI^*}^{MD-ADR} = \sum_{i=1}^{m} \frac{t_i + \tau_i^\top}{P_{i,ADR}} \tag{8.13}$$

The resulting output delay consists of the sum of all operator instance delays d_i plus the processing time t_n of the root operator instance, since this is not covered by the delay d_i. It equals the generic calculation approach:

$$D_{QI^*}^{MD-ADR} = t_n + \sum_{i=1}^{n} d_i \tag{8.14}$$

Note that an arbitrary operator path $OP_j \in QI^*$ can be used, as the delay of all operator paths has been adjusted to be equal. The calculation of the overall memory requirements M also remain unchanged, as proposed by Formula 8.8:

$$M_{QI^*}^{MD-ADR} = \sum_{i=1}^{m}(s_i + B_i) \tag{8.15}$$

Configuration II: Min Delay with Max Data Rate Scheduling

The calculation for combining Min Delay with Max Data Rate scheduling is similar to the previous scheduling configuration. The CPU utilization calculation equals Formula 8.13, as the CPU utilization is independent of the data rate scheduling strategy:

$$C_{QI^*}^{MD-MDR} = \sum_{i=1}^{m}\frac{t_i + \tau_i^{\top}}{P_{i,MDR}} \tag{8.16}$$

The generic formulas for intermediate FIFO memory and intermediate delay calculation can be simplified, since no jitter compensation of $\tau^{\top}$ and $\sigma^{\ominus}$ has to be performed any longer within the intermediate buffer: The FIFO buffer is only required for adapting the buffer access granularities of the producer and consumer operator instance and for holding the data if the producer is too early ($\tau^{\perp}$) or produces too much data ($\sigma^{\oplus}$). Thus, one relies on the jitter-based calculation of Formula 8.3 and sets $\tau^{\top} = 0$ and $\sigma^{\ominus} = 0$ for the delay calculation of a single producer-consumer relationship. One gets

$$d_i{}^{MD-MDR} = t_i + P_{i+1} \tag{8.17}$$

and

$$D_{QI^*}^{MD-MDR} \quad = \quad t_n + \sum_{i=1}^{n}(d_i{}^{MD-MDR}) \tag{8.18}$$

for the total delay, respectively (based on Formulas 8.6 and 8.7). For the FIFO buffer size calculation, the jitters $\tau^{\perp}$ and $\sigma^{\oplus}$ have to be included in the calculation, as the producer operator instance may still jitter in this manner despite the fact that a maximum data rate is assumed. Based on Formula 8.4, one gets

$$B_i{}^{MD-MDR} = \left\lceil bi_{i+1} + \sigma^{\oplus} + r \cdot (t_i + \tau^{\perp}) \right\rceil \tag{8.19}$$

According to Formula 8.8, the total memory requirement is

$$M_{QI^*}^{MD-MDR} = \sum_{i=1}^{m}(s_i + B_i{}^{MD-MDR}) \tag{8.20}$$

Configuration III: Max Throughput with Avg Data Rate Scheduling

Using this scheduling configuration, the time jitter values $\tau^{\top}$ or $\tau^{\oplus}$ do not need to be considered for calculating the CPU utilization. The calculation of D (based on the generic formula 8.9 looks like:

$$C_{QI^*}^{MT-ADR} = \sum_{i=1}^{m} \frac{t_i}{P_{i,ADR}} \tag{8.21}$$

The starting point when obtaining the intermediate and overall memory consumption as well as the intermediate and total delay is the calculation approach from Configuration I. In addition, both the inter-operator delays and the FIFO buffer sizes have to be extended. To account for that, an additional FIFO buffer size B_i^* and an additional delay d_i^* are added for each producer-consumer-relationship. For the concrete calculation of B_i^* and d_i^*, the next section (Section 8.3.2) is recommended due to the complexity of the calculations. Up to now, D has been obtained as

$$D_{QI^*}^{MT-ADR} = t_n + \sum_{i=1}^{n} (d_i + d_i^*) \tag{8.22}$$

based on Formula 8.14. For M, one obtains

$$M_{QI^*}^{MT-ADR} = \sum_{i=1}^{m} (s_i + B_i + B_i^*) \tag{8.23}$$

based on Formula 8.15.

Configuration IV: Max Throughput with Max Data Rate Scheduling

Here, again, the time jitter values $\tau^{\top}$ or $\tau^{\oplus}$ do not need to be considered for calculating the CPU utilization. The calculation of D (directly based on the generic Formula 8.9) looks like:

$$C_{QI^*}^{MT-MDR} = \sum_{i=1}^{m} \frac{t_i}{P_{i,MDR}} \tag{8.24}$$

In order to obtain the intermediate and overall memory consumption as well as the intermediate and total delay, one relies on the calculation of Configuration II, except that–as with the previous configuration–additional delays and FIFO buffer sizes have to be considered. These additional values of B_i^* and d_i^* are the same as above and for their calculation the next section is to be consulted. The inter-operator delay equals the delay of Configuration II (Formula 8.17):

$$d_i{}^{MT-MDR} = t_i + P_{i+1} \tag{8.25}$$

and thus one gets a total delay of

$$D_{QI^*}^{MT-MDR} = t_n + \sum_{i=1}^{n} (d_i{}^{MT-MDR} + d_i^*) \tag{8.26}$$

The intermediate FIFO sizes also equal the ones from Configuration II (Formula 8.19),

$$B_i{}^{MT-MDR} = \left\lceil bi_{i+1} + \sigma^{\oplus} + r \cdot (t_i + \tau^{\perp}) \right\rceil \tag{8.27}$$

and give a total memory requirement of

$$M_{QI^*}^{MT-MDR} = \sum_{i=1}^{m} (s_i + B_i{}^{MT-MDR} + B_i^*) \tag{8.28}$$

8.3.2. JCP+ Extension for the Max Throughput Run Time Strategy

Individual operator instances may be delayed based on the priority within the Max Throughput run time scheduling strategy.

The states of an operator were described in the related work section. The application of that concept on a higher level means that each of the operator instances OI_i of a standing query running with Max Throughput strategy can be in three different states during its 'lifetime:'

1. If OI_i is consuming and producing the desired amount of data periodically, it is in **normal operation state**.
2. If OI_i is blocked due to jitter and compensation of higher-prioritized processes, it neither reads nor writes any data from the input buffer or to the output buffer: it is in the so-called **blocking state**.
3. If OI_i is compensating a jitter, it runs faster than normal, which comes along with consuming and producing data in uninterrupted fashion. This is called the **compensation state**.

Putting the focus on the continuous dataflow between consecutive operator instances, all different constellations of states of the producer and the connected consumer must be considered (Table 8.2). The goal here is to identify the *worst cases* for the required intermediate buffer as well as for the consumer delay to enable continuous data flow. The worst case regarding the required buffer occurs if the producer delivers an extraordinary

amount of data during its compensation and the consumer is blocked (**worst case A**). The *intermediate buffer must be enlarged* to avoid overflow.

In contrast, the worst case regarding the delay time (which the consumer must wait until it starts reading the data from the buffer) occurs if the producer is blocked and the consumer reads an extraordinary amount of data during its compensation (**worst case B**). The *delay as well as the intermediate buffer must be enlarged* to ensure that always enough data are available.

The following two paragraphs derive formulas for the additional amount of required resources. It is assumed that the two consecutive operator instances are OI_i and OI_{i+1}. The associated priorities are p_i and p_{i+1}, respectively. Furthermore, the overall jitter $\tau_{all,i}$ as well as jitter compensation times $t_{comp,i}$ of the involved operator instances are known.

	Producer Operator OI_i		
	normal operation	blocking	compensation
Consumer Operator OI_{i+1}			
normal operation	no additional resources required as both operators work as scheduled with JCP+	if $p_i > p_{i+1}$: impossible if $p_i < p_{i+1}$: covered by **B**	if $p_i > p_{i+1}$: impossible if $p_i < p_{i+1}$: covered by **A**
blocking	if $p_i > p_{i+1}$: covered by **A** if $p_i < p_{i+1}$: impossible	no additional resources required as none of the two operators works; if $p_i < p_{i+1}$: impossible	if $p_i > p_{i+1}$: **worst case A** (too much data) if $p_i < p_{i+1}$: impossible
compensation	if $p_i > p_{i+1}$: covered by **A** if $p_i < p_{i+1}$: impossible	if $p_i > p_{i+1}$: impossible; if $p_i < p_{i+1}$: **worst case B** (data is missing)	impossible combination as only one of the operators can be in compensation state at a time

Table 8.2.: Combination of Operator States

Worst Case A: Producer Must Not Be Blocked

In worst case A, the producer priority is higher than the consumer priority: $p_i > p_{i+1}$. The intermediate buffer must be large enough to additionally hold producer data during the producer compensation process and during the producer's consecutive regular work. The issue is illustrated in Figure 8.24: The producer operator receives a jitter $\tau_{all,producer}$ and adjacently compensates this jitter. During its compensation time $t_{comp,producer}$, the operator fills up the intermediate buffer more and more but–due to the lower priority– the consumer operator is still blocked and thus cannot read out any buffer data. A buffer increment is therefore required. After the compensation, the producer operator continues with its regular periodic work.

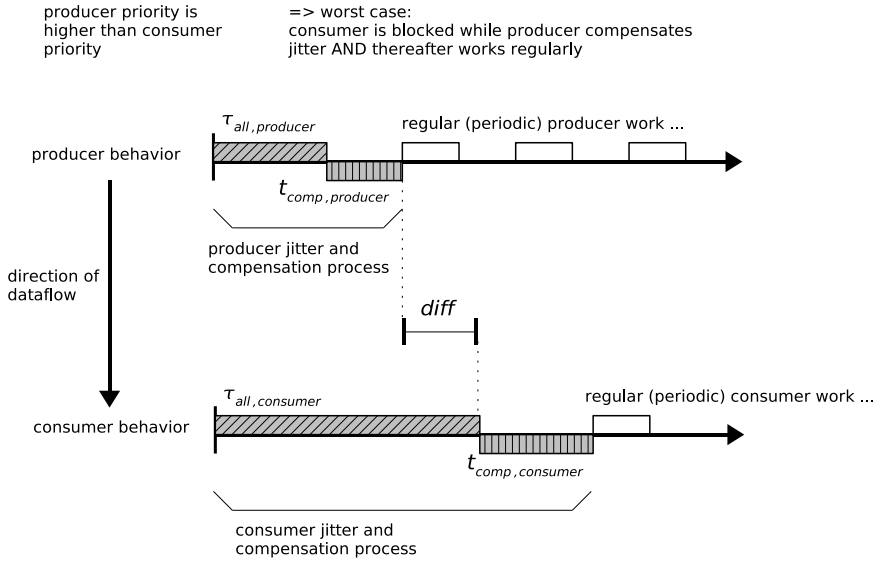


Figure 8.24.: Operator Blocking Behavior: Producer Operator with Higher Priority than Consumer Operator

If the consumer is the direct successor of the producer in order of priorities, the jitter $\tau_{all,consumer}$ of the consumer ends exactly at the time the producer has finished its jitter compensation ($diff = 0$). If there are other operator instances in between of the producer and the consumer regarding the priority, these other operators may compensate before the consumer and $diff > 0$. A further buffer incrementation is required.

The additional buffer requirements (denoted as B_i^*) are given in Formula 8.29. The labels 'producer' and 'consumer' together with the operator dataflow position 'i' and '$i+1$' are used in the formula to ease readability.

The two components of the increased buffer are marked as (a) and (b), respectively. The value of $diff$ can be obtained as

$$diff = \tau_{all,consumer(i+1)} - \tau_{all,producer(i)} - t_{comp,producer(i)}$$

The amount of additionally produced tuples during the producer's compensation is determined by dividing the producer's compensation time by its regular processing time (per run) and multiplying this value with the producer's output batch size (first summand). The second summand incorporates the amount of tuples produced during the time *diff*.

$$B_i^* = \overbrace{\left\lceil \frac{t_{comp,producer(i)}}{t_{producer(i)}} \right\rceil \cdot bo_{producer(i)}}^{\text{(a) producer jitter compensation}} + \overbrace{\left\lceil \frac{diff}{P_{producer(i)}} \right\rceil \cdot bo_{producer,i}}^{\text{(b) producer regular work}} \tag{8.29}$$

A starvation of the consumer is barred as the producer may only produce too much data. Thus a lead time extension is not required in this case:

$$d_i^* = 0 \tag{8.30}$$

Worst Case B: Consumer Must Not Be Blocked

A starvation of the consumer may only happen if the producer priority is smaller than the consumer priority: $p_i < p_{i+1}$. The delay as well as the buffer between the operator instances OI_i and OI_{i+1} must be increased to give the producer enough time for producing as much data as the consumer needs.

This issue is illustrated in Figure 8.25. The consumer operator receives a jitter $\tau_{all,consumer}$ and immediately compensates this jitter. During its compensation time $t_{comp,consumer}$, the operator continuously reads out tuples from the intermediate buffer but–due to its lower priority–the producer operator is still blocked and thus cannot produce any buffer data. A lead time increment (which comes along with a buffer increment) is therefore required. After the compensation, the consumer operator continues with its regular periodic work.

If the producer is the direct successor of the consumer in order of priorities, its overall jitter $\tau_{all,consumer}$ ends exactly at the time the consumer has finished its jitter compensation and it holds that $diff = 0$. If there are other operator instances in between of the producer and the consumer regarding the priority, these other operators may compensate in between and it holds that $diff > 0$.

The additional buffer requirements (denoted as B_i) are given in Formula 8.31. The two components of an increased buffer are again marked as (a) and (b), respectively. The value of $diff$ can be calculated as

$$diff = \tau_{all,producer(i)} - \tau_{all,consumer(i+1)} - t_{comp,consumer(i+1)}$$

The former summand incorporates dividing the compensation time by the regular runtime to determine the number of runs the compensation outlasts. Then, the summand is multiplied with the number of tuples which arrive during each run. The latter summand consists of the amount of additionally required tuples during the time *diff*.

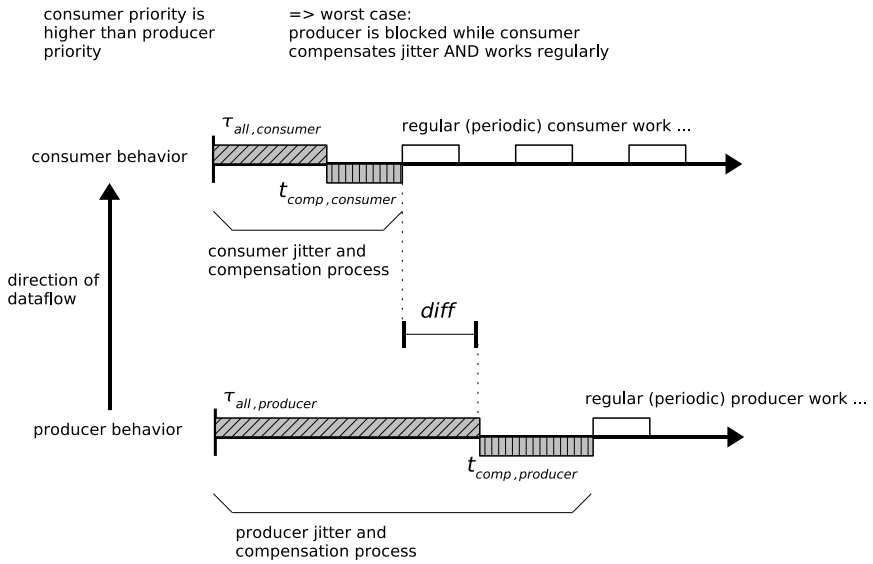


Figure 8.25.: Operator Blocking Behavior: Consumer Operator with Higher Priority than Producer Operator

$$B_i^* = \overbrace{\left\lceil \frac{t_{comp,consumer(i+1)}}{t_{consumer(i+1)}} \right\rceil \cdot bi_{consumer(i+1)}}^{\text{(a) consumer jitter compensation}} + \overbrace{\left\lceil \frac{diff}{P_{consumer(i+1)}} \right\rceil \cdot bi_{consumer(i+1)}}^{\text{(b) consumer regular work}} \quad (8.31)$$

Formula 8.32 calculates the additional lead time requirements (denoted as d_i). There, the additional buffer size B_i^* is divided by the output batch size $bo_{producer}$ of the producer operator to determine the number of periods required to produce that data. Finally, this number of producer periods is converted into a time measure by multiplying the intermediate result with the producer's period length $P_{producer}$.

$$d_i^* = \frac{B_i^*}{bo_{producer(i)}} \cdot P_{producer(i)} \quad (8.32)$$

Now, the final resource (overall memory) and QoS (total delay) for the Max Throughput strategy can be obtained easily by including the results in Formulas 8.22 and 8.23 (Configuration III) and in Formulas 8.26 and 8.28 (Configuration IV).

8.3.3. Overall Resource Calculation and QoS Negotiation Steps

This section summarizes the resource calculation and QoS negotiation steps of QStream. The procedure is illustrated in Figure 8.26. The points where required resources and

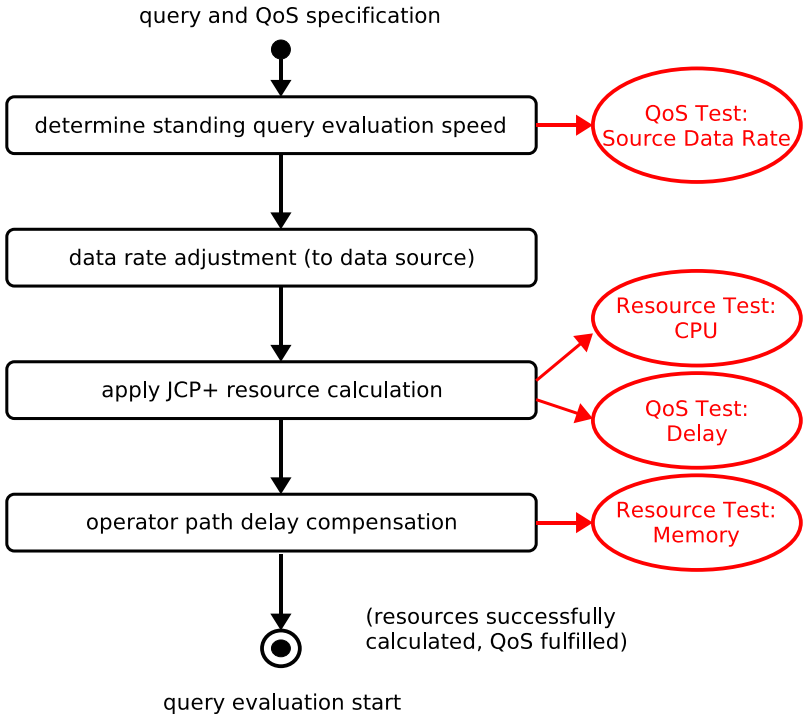


Figure 8.26.: QStream Overall Resource Calculation Steps

QoS are tested against available resources and against user requirements, respectively, are denoted by an ellipse.

The procedure's input parameters comprise the standing query instance QI, the time-based QoS requirements, the data rate of the data source R_{source}, and the selected scheduling configuration. The content-based QoS requirements are currently not considered for negotiation within QStream.

The first step consists of the determination of the standing query instance's evaluation speed by calculating the period length of each operator instance (Section 8.1.1). If a non-controllable data source is connected and the source data rate is not high enough, the standing query is rejected for that reason. Otherwise, if the source data rate is higher than required, an adjustment operator is inserted in the second step. Besides, priorities are assigned to the operator instances in reverse order of their period lengths (following the RMS strategy).

Within the third step, the JCP+ calculation is applied to the previously adapted standing query instance, depending on the selected scheduling strategy for runtime as well as for data rate. Afterwards, the overall delay can be tested against the delay QoS requirement. Furthermore, required processing time resources (percent of CPU utilization) are tested. If one of them is not fulfilled, the standing query is rejected again.

If join operators are involved and the delay of the individual operator paths differ, a delay compensation is applied as the fourth step. There, the FIFO buffers of the respective join input streams with the shorter path delay are extended to hold as many additional tuples as the join input stream delivers during the delay difference. After the join input buffers have been enlarged, the required memory must be tested against the available memory and–if the system's resources are not sufficient–the standing query is rejected.

If all required resources are available and the user-given QoS can be fulfilled, the procedure's results are the parameters for setting up the RMS scheduling. Otherwise, lower requirements in terms of output data rate R_{min} or a higher tolerance regarding the output delay D_{max} must be negotiated and the procedure may be repeated with the new (lower) requirements. If one still does not come to a conclusion, the standing query must be rejected.

8.4. Summary

This chapter presented the main concepts of providing time-based QoS guarantees. The JCP+ calculation approach allows for obtaining operator-based as well as stream-based resources. The scheduling strategies for the individual operator run times as well as for the data rate illustrate different possibilities of mapping the operator-level resource requirements to the system-specific resource allocation interface. The appropriate scheduling strategies can be chosen depending on the available resources and on the user's preferences. Thus, the conceptual basis for calculating, allocating, and assuring resources during runtime is given.

9. The QStream Robustness Concept

At runtime of a QoS-guarantee DSMS, specific attention must be paid to the operator network's behavior to permanently fulfill the QoS guarantees. The general problem is that the resource reservation was based on query and data stream statistics which were supposed to not change over time. Within realistic application scenarios, this assumption does not hold. First, statistics of incoming data can only be estimated based on historical data and on the experience of the DSMS administrator. Second, statistics will change over time along with the input data streams.

QStream's solution for changing statistics is the *robustness concept*, which was introduced in [SLSL05]. It allows to trade off the amount of resources for considering micro jitter and the number of *adaptations* which a DSMS has to perform during runtime. An adaptation is initiated every time the continuous data flow within the DSMS is interrupted due to insufficient resources. From a conceptual poin of view, an adaptation is triggered every time the actual jitter exceeds the amount of micro jitter.

The remainder of this chapter is structured as follows: First, Section 9.1 defines robustness formally. Then, Section 9.2 gives an overview of QStream's adaptation procedure. Thereafter, Section 9.3 introduces the *Data Stream Characteristics (DSCs)*. They are used to describe data streams as well as operator instances. It is assumed that–although in a broader sense–the behavior of operator instances also depends on the input data and is therefore covered by the term 'DSC.' Finally, different prediction models which are to be applied to the DSCs are presented in Section 9.4. The goal there is to obtain JCP+ parameters for the future DSMS runtime.

9.1. Robustness Calculation

Figure 9.1 illustrates the adaptation process: if the data exchange of the DSMS is blocked, an adaptation is triggered. For deriving new scheduling parameters, collected statistics are used together with an appropriate prediction model. The QoS negotiation process and the resource calculation have to be performed anew, the latest resource requirements have to be reserved, and running operator instances have to be re-initialized.

The borderline between micro and macro jitter is determined by the granted resources, and thus, the system's robustness can be adjusted by the DSMS administrator: The more resources are available for considering micro jitter, the less often a DSMS adaptation is required and the higher the robustenss will be.

The respective characteristics of a DSMS running a set of standing queries is reflected by the *robustness curve* which is schematically shown in Figure 9.2. The amount of granted resources for jitter compensation is annotated on the x-axis and the resulting

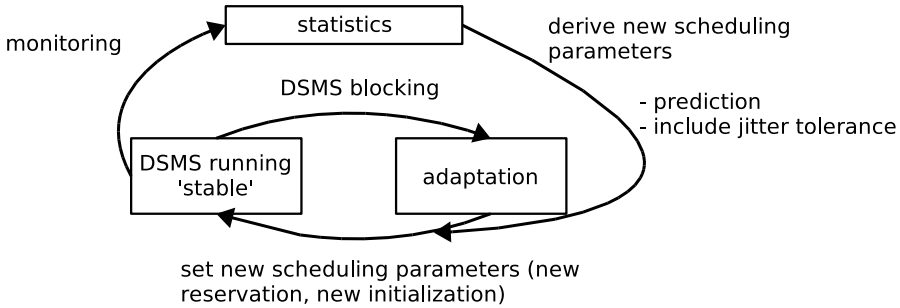


Figure 9.1.: QStream Adaptation Loop

robustness value is annotated on the y-axis. The robustness curve does not necessarily increase monotonically; depending on the time point at which the adaptations take place, the robustness value may remain constant or even decrease temporarily if the resources are slightly increased. This issue will be discussed later in the evaluation chapter.

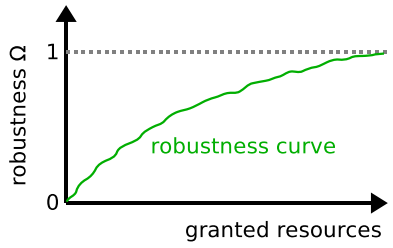


Figure 9.2.: Robustness Curve

The robustness value itself can be derived by counting the number of adaptations per time. It is the ratio between the number of adaptations n and the monitor duration $t_{monitor}$ scaled to an interval of $(0, 1]$ (Formula 9.1). If no adaptation within the considered time span took place, the robustness has a maximum value of $\Omega = 1$:

$$\Omega = \frac{1}{\frac{n}{t_{monitor}} + 1} \tag{9.1}$$

The robustness curve of Figure 9.2 can be 'recorded' by determining Ω for different amounts of granted resources (in terms of jitter tolerance).

9.2. The Macro Jitter Adaptation Concept

The robustenss value Ω directly depends on the number of DSMS adaptations within a given time span. Therefore, the discussion of the remainder of this chapter focuses on adaptation event handling and on the DSMS parameter calculation.

9.2.1. Adaptation Procedure

The adaptation procedure is based on a standing query instance QI and on the DSCs which reside in the DSC repository. The adaptation is initiated by a trigger and may occur at any position of the query instance at any time (Figure 9.3). The triggers considered by QStream are either buffer-full events (e_{full}) or buffer-empty events (e_{empty}) and may be signaled during the read or write operation of an operator instance. The exception is the last operator instance of the query; it may only fire e_{empty} triggers on unsuccessful read attempts.

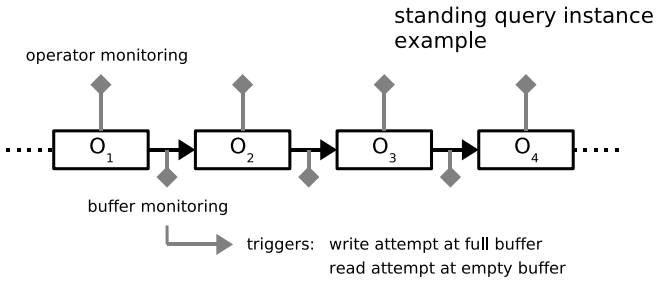


Figure 9.3.: QStream Monitoring Concept - Adaptation Triggers

Identifying the DSMS Bottleneck

When a trigger is fired, the reason for the adaptation can be narrowed down to a specific structural part of the network, which either produces too much or too few data. Note that one must distinguish based on the data rate scheduling strategy which is currently used.

If the DSMS runs on *average data rate strategy*, a trigger is fired every time an operator instance (consumer) attempts to read from an empty buffer and every time an operator instance (producer) tries to write to a full buffer. Otherwise, if the *maximum data rate strategy* is used, only write attempts to full buffers are treated as error events and thus are used as triggers for an adaptation.

The reasons for the buffer read or write faults are twofold:

- If e_{full} was signaled from operator instance OI_i, either one of the operator instances downstream of OI is blocked or the operator instance OI_i itself has produced too much data.

  In the former case, some of the downstream operators, including OI_{i+1}, are unable to consume the desired amount of data. The exact position of the erroneous operator instance within the standing query cannot be identified in an easy way. Any of the downstream operators may cause full buffers but only the e_{full} event which arrives first is considered.

If OI_i itself has produced too much data, the data rate reduction $\frac{bo_i}{bi_i}$ becomes higher than expected, for example, due to a changing value distribution within the input data stream.

- If e_{empty} was signaled from operator instance OI_i, then any operator instance upstream from OI_i may have produced less than the desired amount of data. The erroneous operator instance may be OI_{i-1} or any of its predecessors; in the latter case, the problem of too few data in the intermediate buffers may occur also at other FIFO queues. Similar to signaling the e_{full} events, only the event which arrives first is considered.

Adaptation Procedure

Due to the uncertainty of identifying erroneous operator instances and–much more important–due to the necessity to perform a new data rate propagation, resource and QoS calculation anyway, the adaptation procedure of QStream is kept very simple. It consists of six steps:

1. First and continuously, DSCs are collected and stored in the DSC repository.
2. Then, every time an adaptation trigger (e_{full} or e_{empty}) is fired, the prediction model and the calculation rules for obtaining the required JCP+ parameters for the future DSMS runtime are determined.
3. Based on the new parameters, a new resource reservation following the JCP+ calculation as well as a new QoS negotiation is performed.
4. If the resource reservation and QoS negotiation for QI have been successful, all operator instances belonging to QI are suspended (their work is stopped without deleting or destroying the operator instance).
5. The new period length P_i of each operator instance $OI_i \in QI$ is scheduled and the intermediate FIFO buffers are adapted regarding size. (The minimum FIFO buffer size is given by the FIFO fill level - no tuples are deleted)
6. Finally, the previously suspended operators are re-activated to continue their work with regard to the new delay value which was calculated in between of each two operator instances.

It is important to notice, that the states of all operators and the tuples residing in the FIFO buffers are kept during the adaptation procedure.

9.2.2. Adaptation Effects on QoS and Resources

The success of an adaptation depends on two things: First, there must be enough resources available to put the new DSMS configuration into effect. Second and inextricably linked to that, the (initial) QoS negotiation must be generous enough to allow for the

new (time-dependent) result quality descriptor $\hat{Q}_{time}(R_{min}, D_{max})$. In principle, if not enough resources in terms of memory or processing time are available for evaluating the current standing query, the QoS requirements must be weakened with the following effects:

- If the result data rate requirement R_{min} is reduced, the operator instances are allowed to work slowlier (with larger period lengths), which also leads to larger output delay D. From the viewpoint of resources, the CPU utilization decreases proportionally with a reduced data rate.
- If no compromise for the result data rate is allowed but the output delay can be decreased arbitrarily, one can use the Max Throughput instead of the Min Delay strategy. As a precondition, enough FIFO queue memory must be available.
- If plentiful CPU resources are available, the query may be evaluated using the Max Data Rate strategy. It consumes a lot of CPU time but results in very small intermediate buffers and a low output delay. If–in addition–Min Delay was selected, the focus is on minimal output delay, whereas with Max Throughput strategy, the focus is shifted to minimal buffers.

The resource consumption increases with higher jitter tolerance. Taking this point into account, the DSMS administrator can trade off jitter tolerance and DSMS robustness using the robustness curve. If all efforts of QoS negotiation and resource reservation fail and thus, the QoS of the query result would be too low, the standing query instance can be removed from the DSMS.

9.3. Collecting Data Stream Characteristics

QStream requires DSCs mainly for two reasons: It must be possible to detect the point in time at which an adaptation should be initiated (trigger) and the DSCs must allow the calculation of new scheduling parameters.

DSCs can be either *direct* or *indirect*: Direct DSCs contain low-level information on the data exchange process. Examples are the amount of exchanged data or the operator instances' processing times and selectivity. In contrast, indirect DSCs contain more high-level information. This includes histograms, quantile estimators or stream periodicities and periodic patterns. The current QStream prototype focuses on direct DSCs because they are adequate and sufficient for obtaining JCP+ parameters.

DSCs may be valid and useful either for the current standing query instance only (where they have been measured) or they may be extended to other (concurrently running or future) standing query instances as well. Generally, DSCs obtained directly from the data source stream can easily be used in a variety of standing query instances. Otherwise, if a DSC is measured after some operator instances have pre-processed the data stream, the scope is limited: The DSCs obtained that way can only be 'transferred' to other query instances which are structurally identical up to the point where the DSCs have been measured.

9.3.1. Conceptual DSC Monitoring Architecture

Monitoring and collecting DSCs is a challenging task: on the one hand, the monitoring process must allow for obtaining all DSCs which are required for the adaptation process and the JCP+ calculation. In the case of QStream, this includes information about the established buffers as well as operator instance characteristics. On the other hand, monitoring and collecting must be performed with only a minimal overhead at DSMS runtime to avoid falsification of the measured values and to not exert too much influence on the query evaluation process.

The DSC acquisition task involves two components: a *DSC collector* and a *DSC repository*. The collector is responsible for measuring the characteristics, whereas the repository must offer capacity for storing historical DSC values and functionality for predicting and querying DSCs. In the remainder of this section, three generic DSC acquisition approaches are described. When doing so, the DSC collector is always established as an independent module, whereas the realization and position of the DSC collector (monitor) varies.

Dedicated Monitor Operators

Within this approach, the monitor is a separate DSMS operator instance plugged in between two regular operator instances of the standing query instance (Figure 9.4).

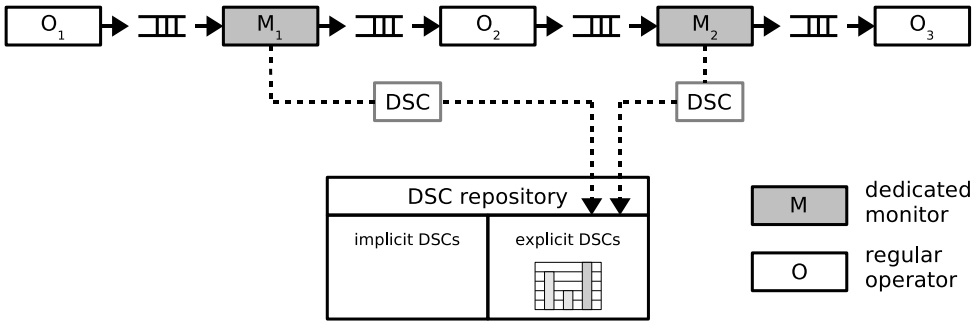


Figure 9.4.: Dedicated Monitoring Concept

The monitor components gather the necessary statistics and hand over the stream tuples unchanged. As a benefit, the DSMS operator concept (and implementational framework) can easily be used for implementing monitor operators. The drawback of this solution is that the query instance must be structurally changed due to the new operator instances. This would also require a new resource calculation and reservation as well as a new QoS negotiation. Furthermore, only indirect DSCs can be monitored, as the monitor has no access to the runtime statistics of other operator instances.

Decoupled Monitors

The second possibility is to run the monitor independent of the standing query instance (Figure 9.5). This leads to a kind of a *shadow network*, as monitors are established for each buffer and for each operator instance which is of interest from this viewpoint.

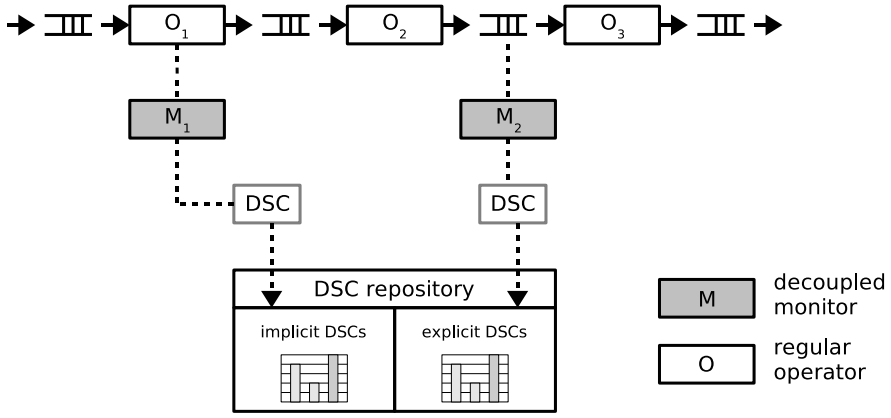


Figure 9.5.: Decoupled Monitoring Concept

This approach is less invasive than the previous one. It allows to dynamically attach and detach monitors to standing query instance components. Furthermore, the monitors can be lower-prioritized than ordinary operator instances, and thus, they do not necessarily influence the runtime behavior of the standing query instance. Consequently–if the DSMS workload is high–it may happen that some DSCs cannot be measured and stored in the DSC repository. Anyhow, this solution has an additional drawback: if extra components for monitoring each buffer and each operator instance are created, the overhead in terms of memory as well as processing time requirements is too high.

Inline Monitors

A combination of the two former approaches was found to be most promising for the use within QStream: a monitor is closely coupled with each of the operator instances (Figure 9.6). This causes only limited overhead and gives flexibility for measuring all required (direct and indirect) operator characteristics. The buffers can be indirectly monitored by plugging some monitor functionality into the I/O layer of the appropriate producer and consumer operators because each intermediate buffer is filled and depleted by exactly one producer or consumer. Most of the operator functionality is implemented as an extra thread running in parallel to each operator instance's main thread. This reduces the influence on the monitored standing query instance to a minimum. In addition, this concept allows for giving the monitor thread a lower priority to not disturb the query evaluation process. Obviously, it also leads to a (tolerable) loss of monitor information in overload situations.

Using the inline monitors, QStream focuses on direct DSCs with the scope of the running query instance. A distinction is made between *error events* and *re-estimators*.

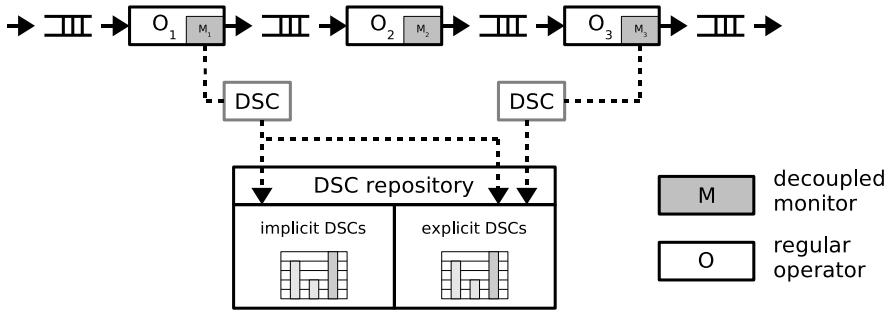


Figure 9.6.: Integrated Monitoring Concept

Buffer overflow and buffer underrun events belong to the error events and are used as triggers for adaptations. In comparison, concrete re-estimators are

- **Average operator instances processing time**: For obtaining the processing times, the monitor uses the information provided by the operating system's scheduler. The scheduler stores the cumulated number of time slices (so-called 'jiffies') which an operator instance has used. If this value is multiplied with the duration of a such a time slice (CPU clock-dependent), one obtains an operator instance's processing time. For JCP+, the average processing time t is of interest; it can easily be obtained by applying an average calculation to the individual values.
- **Processing time jitter (absolute and cumulated)**: To describe the processing characteristic more precisely, a processing time jitter in terms of absolute minimum (negative) $\tau^{\perp}$, absolute maximum (positive) $\tau^{\top}$ and cumulated maximum (positive) $\tau^{\oplus}$ is calculated from the individual processing time measurements and stored in the DSC repository.
- **Average number of produced tuples**: The average number of output tuples per operator run *bo* is obtained by monitoring the write actions to an operator's outgoing FIFO queue. Thereafter, the average is calculated.
- **Output size jitter (cumulated)**: The maximum cumulated (positive) and the minimum cumulated (negative) output size jitters $\sigma^{\oplus}$ and $\sigma^{\ominus}$ respectively are determined and stored in the DSC repository in the same manner as the processing time jitter.

9.3.2. DSC Measurement and Collection Concepts

The individual transmission of each processing time value and each output size value from the inline monitor to the DSC repository, obviously, would cause a non-tolerable overhead regarding the runtime of the monitor procedure. In the worst case, more statistical information than stream tuples would have to be handled by the DSMS.

For that reason, QStream pre-aggregates the operator instance statistics t and *bo* within the time span defined by a so-called *basic window* at the inline monitor. Every

time the basic window has elapsed, the aggregated characteristics are transferred to the DSC repository. Figure 9.7 depicts this aspect and denotes the basic windows as $w_1, ..., w_i, ..., w_n$.

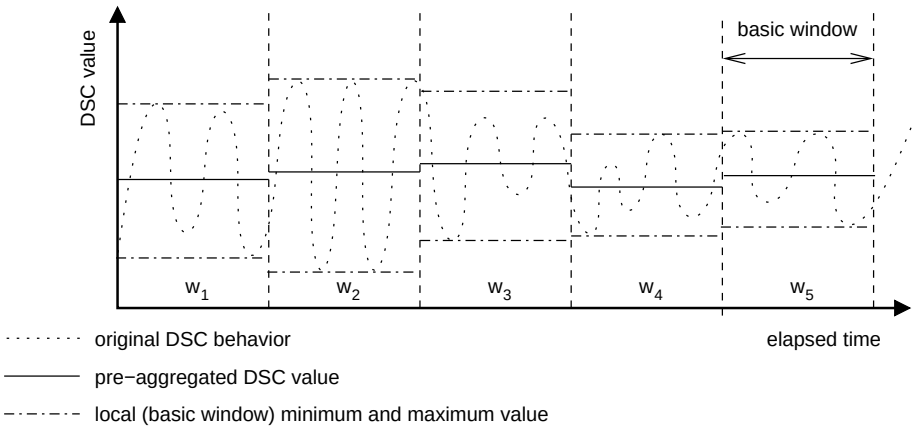


Figure 9.7.: Statistics Pre-Aggregation within an Operator Instance

The pre-aggregation causes problems with obtaining the overall operator instance's jitter characteristics: the absolute and cumulated minima and maxima cannot be obtained directly, as the individual measured values of t and bo are not available any longer within the DSC repository. Instead, the DCSs must be pre-calculated separately for each basic window, transferred to the DSC repository, and the final (global) DSCs have to be determined on the basis of the local ones. Unfortunately this is only possible for the average values and for the *absolute* minima and maxima–not for the cumulated ones. Thus, if the two-step aggregation concept is used, $\tau^{\perp}$ and $\tau^{\top}$ can be directly calculated, whereas $\tau^{\oplus}$, $\sigma^{\ominus}$ and $\sigma^{\oplus}$ can only be estimated based on the previous jitter values.

Inline Monitor Synchronization

The inline monitor is a component running in parallel to the operator instance at a lower priority. The data are transferred asynchronously by the inline monitor at the end of each basic window. The synchronization between operator instance and inline monitor is illustrated in Figure 9.8. The operator instance works as long as the basic window lasts and stores the characteristics into a ring buffer. Then, it triggers the collection and transfer process. If the inline monitor receives that signal, it creates an appropriate DSC packet, makes an additional test for its validity (i.e. test if the operator instance has already overwritten parts of the ring buffer) and sends the packet to the repository. The inline monitor works periodically with the same period length as the operator instance to ensure that it can pick up the statistics after each operator instance run (if the basic window would have elapsed).

If the CPU utilization is too high, it is possible that the inline monitor may not receive enough processing time to perform its work. Then, a shadow storage concept for handing over the DSCs from the operator instance to the inline monitor could easily be applied.

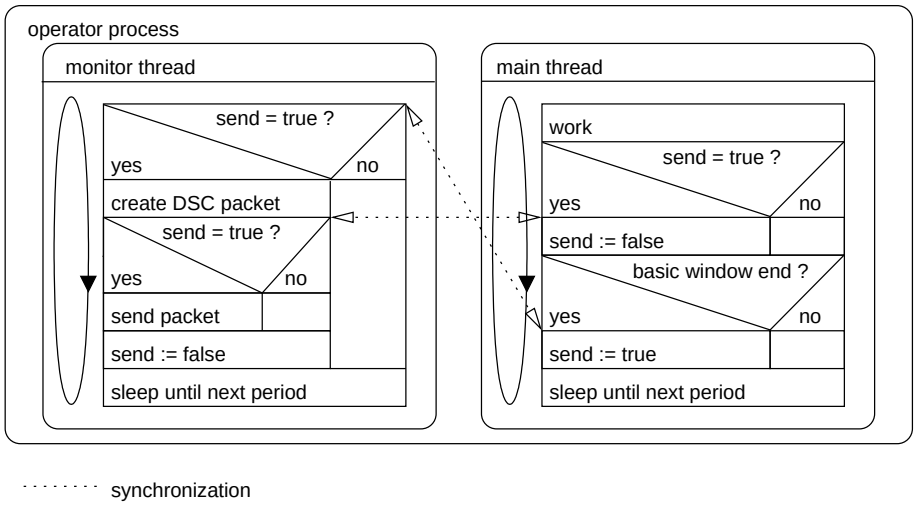


Figure 9.8.: Inline Monitor Synchronization

DSC Repository

The DSC repository stores characteristics of all operator instances. Only a limited history per operator instance can be maintained due to space restrictions. There are two cases which must receive particular attention: First, if an error event was sent in addition to the DSC packet, the DSCs of the current basic window must not be considered for further processing–they can be dropped as they recorded erroneous behavior of the operator instance(s). Second, there may be gaps in the DSC history due to overload situations where characteristics could not be sent at all. Thus, the range of historical DSCs will be restricted to a time span without gaps and error events.

9.4. Prediction Models and DSMS Parameters

The QStream prediction model focuses on the *trend* and on the *period*. The recognition of periodic behavior within the DSCs is restricted to the amount of stored DSC history: if the periods are larger than the time span covered by the repository, they will not be detected by any prediction model. Otherwise, if the DSC periods are shorter than the time span covered by a single basic window, the period will stay unconsidered, too. The following prediction models are well-known and are used as examples within the QStream adaptation framework. They focus on the case where the DCSs behave periodically and the period length is somewhere in between of the basic window length and the time span covered by the DSC repository.

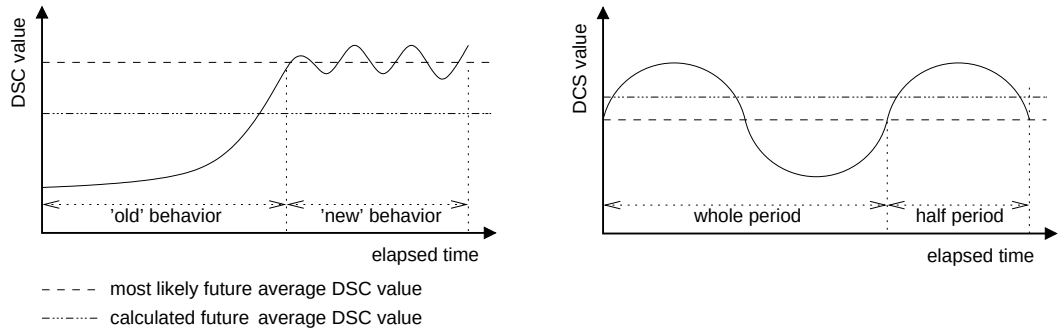
9.4.1. Prediction Models

It is implicitly assumed that the data stream's as well as the operator instances' future behavior is similar to the past behavior. One focuses on the new average value (arith-

metic mean) of the processing time and output size as well as the appropriate jitter characteristics (minima and maxima). In a first step, the prediction model has to identify the historical time span which shall be considered. Secondly, the DSC values of that time span are used to calculate the new arithmetic mean as well as the appropriate jitter. Within QStream, two kinds of prediction models are used. They incorporate either fixed or data-dependent (variable) weighting of the historical data and are described next.

Algorithms with Fixed Weighting

A very simple prediction would be to use the stored arithmetic means over all (historical) basic windows, calculate the average and use this value as the new DSC mean. When doing so, each historical DSC value would receive the same weight. This prediction is very easy to implement and causes only low runtime overhead. This goes on the cost of the prediction quality. Figure 9.9 shows two example DSC histories which cause the model to fail.



(a) trend within DSC behavior (b) periodic DSC behavior

Figure 9.9.: Problems with Equal-Weighting

First, in Figure 9.9(a) the most recent DCS behavior significantly differs from the historical one and the determined average value is clearly not a good choice. In Figure 9.9(b), one DSC period and a half is contained in the DSC repository and due to this (due to the 'half' period), the predicted average becomes imprecise again.

A better suited prediction model is *exponential smoothing*, which gives a stronger weight to the more recent DSC values. The prediction quality is better than with simple equally weighted average calculation as it includes the trend of the DSCs.

Algorithms with Data-Dependent (Variable) Weighting

A more sophisticated prediction (in comparison to the formerly presented models) is to apply a trend analysis to the stored DSC values (based on [Sch01]). The goal is to detect a preferably long periodic behavior and to use this behavior for the prediction of the future arithmetic mean. Figure 9.10 depicts this idea by showing trend lines for different historical ranges.

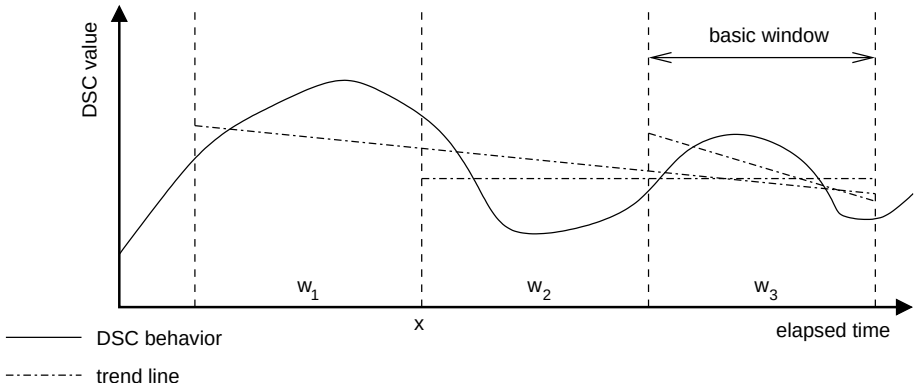


Figure 9.10.: Trend Analysis Approach

In principle, all trend lines ranging from the most recent DSC value back to the complete history are calculated. Then, the trend line with the smallest gradient describes the longest and most recent periodic behavior. It is used for identifying the historical time span of interest, which overspans the windows w_2 and w_3 in the example. In other words: the new average is calculated using the set of DSC values from time point x in history up to the most recent DSC value.

Selecting and Evaluating Prediction Models

All of the described prediction models can be selected within the QStream DSMS. They differ in the complexity (overhead at runtime) and in the prediction quality depending on the input signal. Their 'prediction precision' can be evaluated by applying each of them to the same input data characteristic and comparing the predicted mean values with the actual ones (for example, by calculating the mean square error). With that concept, the prediction model can be replaced arbitrarily at runtime with the goal of the best prediction quality for the appropriate data stream and operator instance behavior. Another possibility would be to choose the most adequate model in an a priori fashion by using additional knowledge of the source data stream or of the operator instance implementation.

9.4.2. Scheduling Parameter Determination

This section describes how to obtain the new JCP+ parameters from the stored DSC values. Within the first part, all required parameters are obtained directly from the individual DSCs. Then, within the second part, the JCP+ parameters (except for cumulated jitter) are calculated from the pre-aggregated DSC values.

Direct Parameter Calculation

In the context of the direct parameter calculation, the basis is the individual operator instance processing time t_j together with the number of produced tuples bo_j which have been gathered for each operator run.

If the prediction function $f_{predict}$ is applied to a number of $(bo_1, ..., bo_j, ..., bo_n)$ output-sized DSC values, it results in the predicted value $bo_{predict}$.

$$bo_{predict} = f_{predict_1}(bo_1, ..., bo_j, ..., bo_n) \tag{9.2}$$

In the same manner, the predicted processing time average $t_{predict}$ can be obtained from the individual operator instance's processing times $(t_1, ..., t_j, ..., t_n)$:

$$t_{predict} = f_{predict_2}(t_1, ..., t_j, ..., t_n) \tag{9.3}$$

The prediction functions $f_{predict_1}$ and $f_{predict_2}$ can be chosen arbitrarily depending on the assumed DSC behavior.

After the prediction of the average runtime and output size, the appropriate jitter values of the past are determined. It is important to notice that this procedure can only result in lower jitter values than before due to the following reason: If the former jitter tolerance was not sufficient and thus an adaptation was triggered, no larger jitter values than assumed by the previous JCP+ calculation can be found in the DSC repository. As a consequence, this procedure is only useful for reducing the jitter tolerance to the required minimum.

If the reason for an adaptation obviously does not lie in a wrong value of bo or t, one must increase the tolerance of processing time or output size jitter. This may be done either based on estimations or stepwise until no more adaptations occur. The procedure of determining decreased jitter values differs for output size and processing time:

- **Output size jitter determination:** For the output size jitter, the cumulated minimum $\sigma^{\ominus}$ and the cumulated maximum $\sigma^{\oplus}$ are calculated based on the single output size values bo_j and based on the new average output size $bo_{predict}$. When doing so, for each bo_j, the signed difference to the average is added to σ. Simultaneously, the overall minimum and maximum of the cumulated jitter are maintained. The procedure is formalized in Algorithm 2.

  The result values bo_{new} as well as $\sigma^{\ominus}$ and $\sigma^{\oplus}$ are directly used as input for performing a new JCP+ calculation.

- **Processing time jitter determination:** The processing time jitter is obtained similar to the output size jitter. The only difference is that the new absolute minimum (negative) processing time jitter $\tau^{\perp}$ and the new absolute maximum (positive) processing time jitter $\tau^{\top}$ must be calculated in addition to the new cumulated maximum (positive) processing time jitter $\tau^{\oplus}$. The absolute minimum

Algorithm 2 Calculation of minimum and maximum cumulated output jitter of an operator instance

Require: scheduled average output sizes $\overline{bo}$, set of single output sizes $bo_1, ..., bo_j, ..., bo_n$

```
 1: # initialization
 2: set σ^⊕ := 0
 3: set σ^⊖ := 0
 4: set σ := 0
 5:
 6: # iteration over all measured values bo_j
 7: for all bo_j from j = 1, ..., n do
 8:
 9:    # calculate new size jitter value
10:    σ  := σ +  (bo_j − bo_new)
11:
12:    # refresh new jitter maxima and minima
13:    if σ > σ^⊕ then
14:       # current run's output is larger than previous maximum
15:       σ^⊕ := σ
16:    else if σ < σ^⊖ then
17:       # current run's output is smaller than previous minimum
18:       σ^⊖ := σ
19:    end if
20: end for
```

and the absolute maximum processing time jitter throughout all stored values are calculated straightforward as:

$$\tau^{\perp} = \max_{j=1}^{n} (t_{new} - t_j) \tag{9.4}$$

$$\tau^{\top} = \max_{j=1}^{n} (t_j - t_{new}) \tag{9.5}$$

The concept of calculating the cumulation jitter maximum is costlier and formalized in Algorithm 3. In contrast to Algorithm 2, the current cumulated jitter τ value must not fall below zero because the (periodically scheduled) operator instances cannot profit if an operator instance finishes earlier than expected, and in contrast to the cumulated output size jitter, the saved amount of time cannot be carried over to future operator runs. Thus, if the current processing time t_j is smaller than the average t_{new}, the cumulated value is decreased but does not become smaller than zero.

As a result of the processing time jitter calculation, the predicted average runtime $t_{predict}$ as well as the jitter values $\tau^{\perp}$, $\tau^{\top}$ and $\tau^{\oplus}$ can be used as input parameters for performing a new JCP+ calculation.

Algorithm 3 Calculation of maximal cumulated processing time jitter of an operator instance

Require: average (old) processing time $\overline{t}$, set of processing times $t_1, ..., t_j, ..., t_n$

```
# initialization
set τ := 0
set τ^⊕ := 0

# iteration over all measured values
for all t_j from j = 1, ..., n do

   # calculate new time jitter value
   if t_j > t_new then
      # add jitter unconditionally
      τ := τ + t_j − t_new
   else
      # delete only a (reduced) amount of jitter
      # that jitter doesn't become negative
      if τ − (t_j − t_new) ≥ 0 then
         # early finish can be used for lowering cumulative jitter
         τ := τ − (t_j − t_new)
      else
         # early finish can be used for setting cumulative max time to zero as a minimum
         # do not allow negative values
         τ := 0
      end if
   end if

   # refresh new maxima
   if τ > τ^⊕ then
      set τ^⊕ := τ
   end if
end for
```

Two-Step Parameter Calculation

If the DSCs are pre-aggregated within the monitor, the calculation rules must be modified. Within a first step, aggregates of $\overline{t}_{w_j}$ and $\overline{bo}_{w_j}$ are calculated within the inline monitor at the end of each basic window w_j using the individual output sizes and the individual processing times, respectively. This is achieved by Formulas 9.6 and 9.7:

$$\overline{bo}_{w_j} = \frac{\sum_{i=1}^{m} bo_i}{m} \tag{9.6}$$

$$\overline{t}_{w_j} = \frac{\sum_{i=1}^{m} t_i}{m} \tag{9.7}$$

Furthermore, the absolute minimum (negative) $t_{w_i}^{\perp}$ as well as the absolute maximum (positive) $t_{w_i}^{\top}$ of the operator instance processing times are calculated per basic window w_i by using Formulas 9.8 and 9.9. The basis is a window's average processing time $\overline{t}_{w_j}$:

$$\tau_{w_j}^{\perp} = \max_{i=1}^{m} (\overline{t}_{w_j} - t_j) \tag{9.8}$$

$$\tau_{w_j}^{\top} = \max_{i=1}^{m} (t_j - \overline{t}_{w_j}) \tag{9.9}$$

All four aggregate values are sent to the DSC repository and stored there.

Then, within the DSC repository, the final average output size $bo_{predict}$ and the final average processing time $t_{predict}$ are predicted similar to the one-step parameter calculation. Therefore, the pre-aggregated average values are inserted into Formulas 9.10 and 9.11.

$$bo_{predict} = f_{predict}(\overline{bo}_{w_1}, ..., \overline{bo}_{w_j}, ..., \overline{bo}_{w_n}) \tag{9.10}$$

$$t_{predict} = f_{predict}(\overline{t}_{w_1}, ..., \overline{t}_{w_j}, ..., \overline{t}_{w_n}) \tag{9.11}$$

The new absolute minimum and the absolute maximum processing jitter $\tau^{\perp}$ and $\tau^{\top}$ can be calculated from the predicted average $t_{predict}$ and the additionally transferred minima and maxima. When doing so, one must consider that the transferred minimum and maximum jitter was only valid within the window w_j. It must now be adjusted using Δ_t, which equals the difference of the basic window average processing time $\overline{t}_{w_j}$ and the predicted new average processing time $t_{predict}$: $\Delta_t = t_{predict} - \overline{t}_{w_j}$. Then, the global extreme values are obtained as

$$\tau^{\perp} = \max_{j=1}^{n} (\tau_{w_j}^{\perp} - \Delta t) \tag{9.12}$$

$$\tau^{\top} = \max_{j=1}^{n} (\tau_{w_j}^{\top} - \Delta t) \tag{9.13}$$

Obtaining the data source input rate: If the connected data source works independent of the DSMS (non-controllable data source), its data dissemination rate may change over time. The DSMS input data rate can only be estimated, as the monitor cannot recognize the write operations to the first buffer: in case that the first buffer does not fill up, the input data rate is lower than assumed. Otherwise, for buffer overflows, the data source dissemination rate was too high for the current DSMS configuration.

9.5. Summary

This section presented the robustness concept of QStream. The motivation was to treat robustness as an additional quality measure which can be influenced by the user in terms of more generous resources. Aside from the monitoring of operator instance and data exchange behavior, the adaptation event itself can be subject of monitoring and evaluation but on a more macroscopic layer. For example, operator instances or operator instance sequences which frequently cause adaptation events can be identified. Furthermore, data sources which show unpredictable behavior as well as operator instances with sporadically increased per-run processing times may receive particular attention.

Furthermore, the robustness concept offers a measure for the effectiveness of a prediction model: the fewer adaptations occur with a constant amount of jitter tolerance (i.e. the higher Ω is), the better do the predicted JCP+ parameters and the jitter estimation meet the DSMS requirements. Thus, different prediction models can be easily compared to each other by calculating the robustness value based on the same standing query instance and based on the same input data (stream) configuration.

Part III.

QStream Prototype and Evaluation

10. The QStream Prototype

This section provides an overview of the QStream DSMS from the implementational perspective. It starts with an introduction of the DSMS application concept (Section 10.1). Thereafter, in Section 10.2, the DSMS components and their implementation are described. Finally, Section 10.3 discusses QStream's interface for acquiring sensor data using dedicated data acquisition hardware.

10.1. Application Concept

The current QStream implementation runs within a centralized computing environment. The coarse-grained components are illustrated in Figure 10.1.

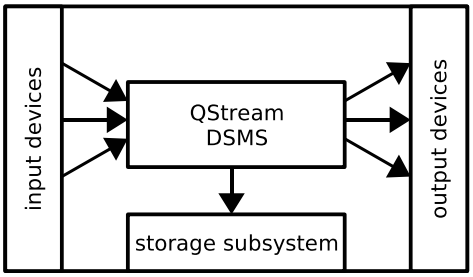


Figure 10.1.: QStream Hardware Environment

QStream favors so-called data acquisition (DAQ) hardware as input devices. A network interface can be used if the appropriate network driver offers real-time capabilities; it is not yet implemented and left out for future work. For testing purposes and for running experiments materialized stream data also residing in the storage subsystem can be used. QStream pre-loads these data within the current implementation, as disk access cannot assure any QoS. Furthermore, the output options only include presenting it on the monitor GUI and writing it back to the storage subsystem.

QStream runs as an application within a Linux environment. The latter is extended to offer real-time functionality by using the Real-time Application Interface (RTAI, [Mou03, Man03, Tea02, Tea06]). Figure 10.2 classifies QStream regarding the different software layers.

Operating System Layer

The operating system layer offers full linux functionality combined with support for *hard* and *soft* real-time applications. Note that a hard real-time program receives higher

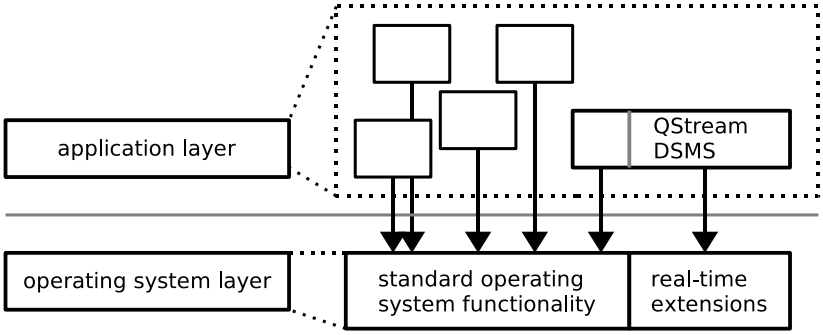


Figure 10.2.: QStream Software Environment

priority than the linux operating system kernel and thus cannot be interrupted by the ordinary kernel work, such as handling input/output of peripheral devices, executing system calls, etc. In contrast, a soft real-time component runs with a priority higher than all other (best-effort) processes but may be interrupted by kernel activity.

During the RTOS initialization time, the basic timer period must be set. This timer period is the smallest time slice after which the scheduler can switch between concurrent operators - it represents a trade-off between fine-grained time allocation on the one hand and scheduling (context switching) overhead on the other hand. Compared to the concept of microperiods, the basic timer period is set only once (during initialization) and is valid for all components to be executed within the RTOS environment. For QStream, a value of $20\mu s$ was selected.

Application Layer

The application layer contains the QStream DSMS as well as other (QStream-independent) program components. The QStream components run as so-called LXRT programs[1] in hard or in soft real-time; both make use of the real-time extensions provided by RTAI. Other applications (besides QStream) are supposed to run in best-effort mode; they only make use of standard Linux functionality. An alternative would be to implement each component as a kernel module and thus to start it with the 'insmod' command (to load/start the module) and to stop it with the 'rmmod' command (to remove/stop the module). This provides a smaller execution overhead but complicates the development process.

10.2. Architecture

QStream consists of the program components, as illustrated in Figure 10.3. The first component is the *query engine* for performing the main data stream evaluation work. The user interface is implemented within the *controller*. It allows for choosing standing queries, negotiating Quality-of-Service, setting up parameters of the RTOS environment

[1] RTAI extension to allow real-time components to run as userspace programs

and configuring the runtime environment. The *statmon* receives the DSCs from the operator instances of the query engine and maintains them using the *DSC repository*. The *monitor GUI* is a graphical monitor for displaying and–if required–for storing arbitrary stream data and DSC information. The *global catalog* is not an active component but a kind of repository for holding DSMS and network configuration information and for 'light-weight' exchange of low-volume data. All components are described in more detail within the remainder of this section.

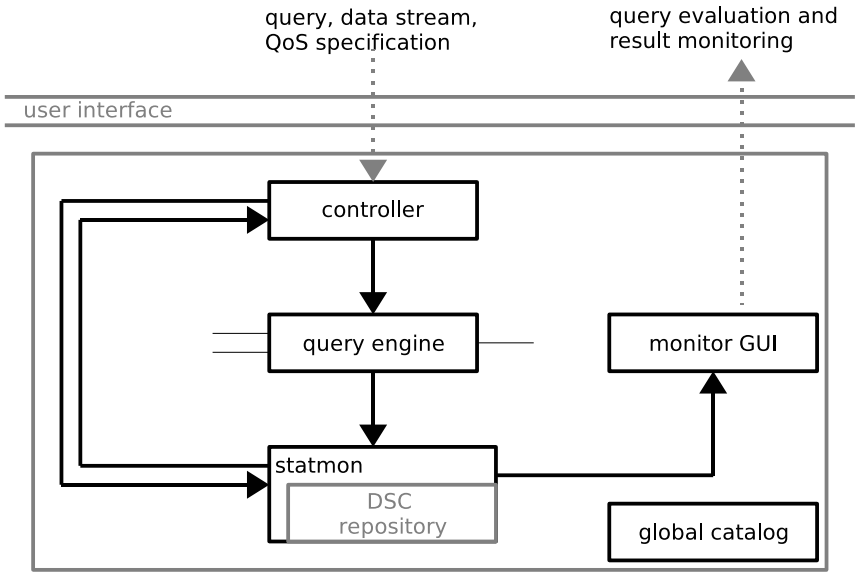


Figure 10.3.: QStream Architecture

The global catalog is implemented on a shared-memory basis. Each QStream component maintains appropriate pointer information to access the catalog entries. Catalog entries can be divided into *global* (DSMS-specific) and *local* (operator- and FIFO queue-specific) entries. The former include monitor and adaptation information like

- length of the basic window maintained by the inline monitors,
- number of basic windows the DSC repository consists of,
- prediction model to be used by the statmon,
- selected data rate and runtime scheduling strategy, and
- update frequency of the monitor GUI.

The operator- and FIFO queue-specific entries are

- operator instance's monitor configuration (switch inline monitoring on/off, select the DSCs to be monitored),

- operator task descriptions (ID of operator task and appropriate inline monitor task),
- operator instance control information (pause and resume operator instances if required),
- FIFO queue descriptors (queue size, read and write pointers for synchronization), and
- stream schema information (number, type and size of attribute values of the intermediate data stream descriptors).

Controller

The controller component is implemented as a soft real-time process. It provides an interface for selecting standing queries as well as for specifying QoS requirements. Furthermore, the scheduling strategy can be configured and the QoS negotiation process can be performed. Most important, the query evaluation process can be started and stopped. During runtime, the controller is informed whether an adaptation trigger is fired. If so, the controller inquires the most recent standing query parameters from the statmon and performs an adaptation.

Query Engine

The query engine (Figure 10.4) is not a monolithic program component; it consists of individual instances of elementary operators which have been introduced previously in Chapter 7. Every two operator instances are connected by a FIFO queue to allow for uni-directional data exchange.

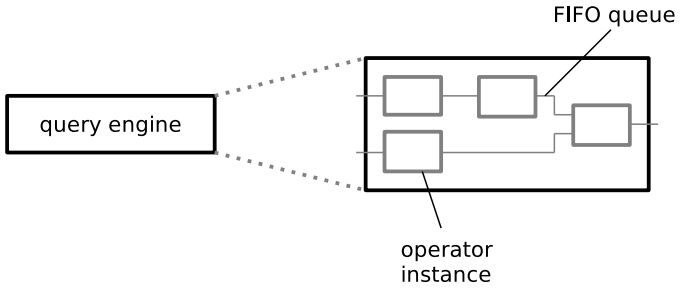


Figure 10.4.: Query Engine Implementation

Regarding the real-time standing query evaluation, each operator instance is an independent RTAI hard real-time component running as an LXRT thread. From a conceptual point of view, the operator instances run in parallel, each of them with its own periodicity. The runtime parameters as well as the input and output stream descriptors are maintained within the global catalog. In addition, at creation time, the operator

instances receive some initialization parameters via the command line interface. During runtime, the global catalog is used (and polled) to receive recent control information (for suspending, resuming or exiting) from the controller.

The inline monitor (thread) contained in each operator instance is responsible for sending all DSCs to the statmon each time a basic window has elapsed. This happens in similar fashion as the regular data exchange via a FIFO queue.

Statmon and Monitor GUI

The statmon as well as the monitor GUI run as soft real-time processes as it is tolerable to lose some DSCs or some monitor GUI input data in situations of high stream load. The statmon maintains a ring buffer as a history of basic window DSCs of each operator instance (→ DSC repository). Furthermore, the statmon applies the selected prediction model on that basis every time it receives an inquiry from the controller. The DSCs are obtained via a FIFO queue.

The monitor GUI is the component of the QStream DSMS with the lowest priority. It receives the relevant monitor data via a dedicated FIFO queue from the statmon. Furthermore, it provides a user-interface for selecting and configuring a variety of views and diagrams and additionally enables storing the results on hard disk. Generally, the monitor GUI is useful for short-term as well as long-term monitoring of processing times, FIFO fill levels, adaptation effects and DSMS robustness.

10.3. Sensor Data Acquisition

The origin of a data stream can be manifold but within this first prototype, QStream favors low-level data sources by supporting data acquisition (DAQ) from sensors measuring physical values. The sensor signals (analog or digital) are sampled[2] using DAQ hardware. Thereafter, they are available as DSMS source streams. Depending on the kind of sensor signal, the stream may have a *continuous*, a *discontinuous* or an *event* character and thus all three partial stream classes CS, DS and ES have to be annotated as meta data, respectively.

The approach of acquiring sensor input data is illustrated in Figure 10.5. First, the sensor 'translates' physical values (temperature, pressure, brightness, strength, etc.) to a voltage value. Then, the voltage value is converted into a data stream tuple by taking the voltage at discrete time points (→ *discretization* or *sampling*), followed by a *quantization* of the value with a certain accuracy. Both steps, the discretization and the quantization, are technically limited by the DAQ hardware. QStream, for example, has been tested with a National Instruments DAQ board (NI 6024E, [Nat04]) offering 12 bit quantization accuracy and a sampling rate/frequency of $200,000 \frac{samples}{second}$.

[2] The sampling process at this point has nothing to do with database or data stream sampling methods - its origin lies in the signal processing techniques and in the analog-to-digital conversion (ADC)

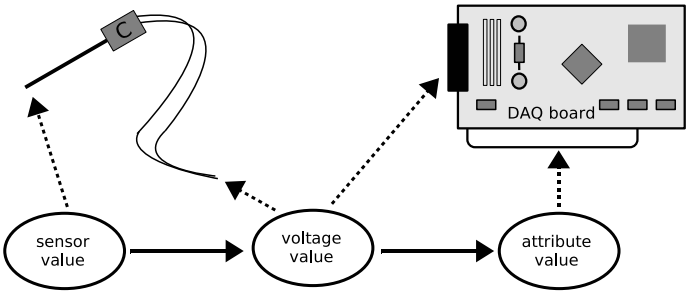


Figure 10.5.: DAQ Hardware Components

10.3.1. The Comedi Device Interface

Due to the fact that a variety of DAQ hardware with different capabilities regarding sampling rate, quantization accuracy and input voltage sensitivity is available, it is not beneficial to implement specific hardware support for each individual device. Therefore, QStream makes use of the *Comedi* interface ([SHB03]). Comedi stands for 'linux control and measurement device interface'. It encapsulates DAQ hardware and provides an API for accessing it from linux applications. Comedi allows to access different (real) input channels as well as other on-board DAQ hardware (e.g. timers, counters, configuration information) through so-called 'subdevices' of the actual device (Figure 10.6). For example, counters and timers can be programmed and used for measurement tasks and configuration subdevices are–as the name already indicates–required for Comedi configuration and hardware initialization.

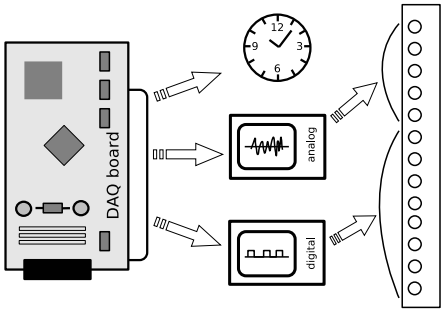


Figure 10.6.: Subdevices Provided by Comedi

The Comedi concept has been adapted to the RTAI real-time environment, which allows applications to acquire sensor data in real-time, process them in real-time and output the result in real-time, too. Figure 10.7 illustrates the different components involved in the DAQ process: For each DAQ hardware component, a vendor-specific driver module exists. On top of it, the 'kcomedilib' provides an API for kernel modules - either for real-time or for non-real-time. User-mode applications are supported via two libraries: 'comedilib' for non-real-time applications and 'libcomedilxrt' for real-time applications (also called *LXRT-Comedi*). QStream makes use of the LXRT-Comedi

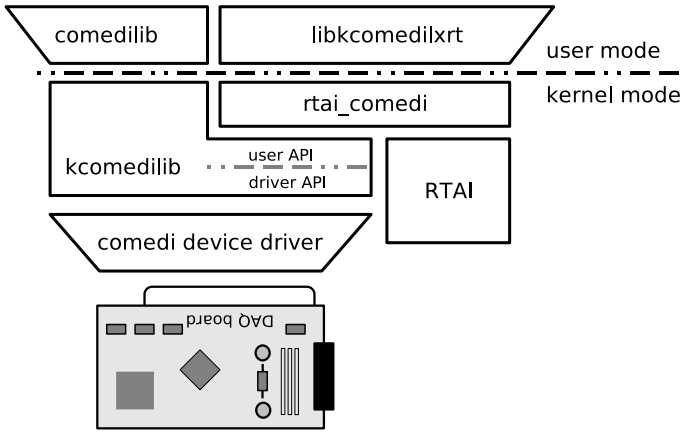


Figure 10.7.: DAQ software stack

API for accessing the Comedi services as it is an user-mode application. The basic programming concepts as well as an API documentation of Comedi are given in [SHB03].

10.3.2. QStream Data Acquisition Techniques

Most of the data acquisition hardware provides the capability to acquire the data in either *synchronous* or *asynchronous* mode. The difference lies in how far the DAQ procedure is pushed down to the hardware: Reading data from the sensors has to be performed repeatedly, as the acquisition should result in a data stream. In synchronous mode, the consecutive acquisition steps are controlled by the application program, whereas in asynchronous mode, the hardware of the DAQ board takes over this job. Both acquisition techniques are described next.

Synchronous Data Acquisition

In synchronous mode, the application program itself initiates every single sensor reading. Then, the application has to wait until the results are available and thus it is blocked during that time. This procedure is sketched in Figure 10.8. The user program instructs

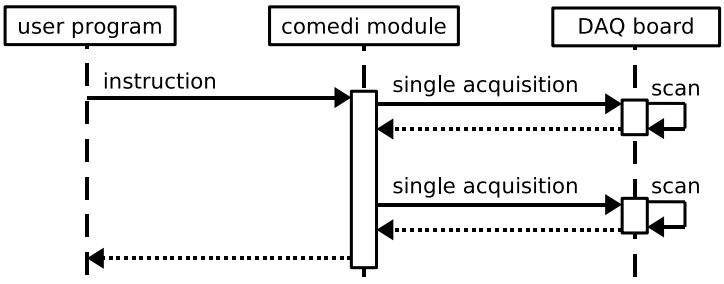


Figure 10.8.: Synchronous Data Acquisition Sequence

the Comedi module to either acquire a single sensor value or a (finite) sequence of sensor values from one or from multiple sensors. Thereafter, the Comedi module forwards the request to the DAQ hardware and if the results become available, the Comedi module returns them to the calling application. Due to the blocking behavior, this technique is not suited for acquiring data streams.

Asynchronous Data Acquisition

If precisely timed equidistant sensor readings are indispensable, the data acquisition procedure has to be pushed down to the DAQ hardware as far as possible. In asynchronous DAQ mode, repeated and periodic readings from one or from multiple sensors can be programmed and scheduled. In contrast to the synchronous mode, the timing of the readings are controlled by the DAQ hardware and–once the acquisition process has been initiated–the acquisition results can be obtained continuously by the application program (the calling program component is not blocked). The procedure is illustrated in Figure 10.9.

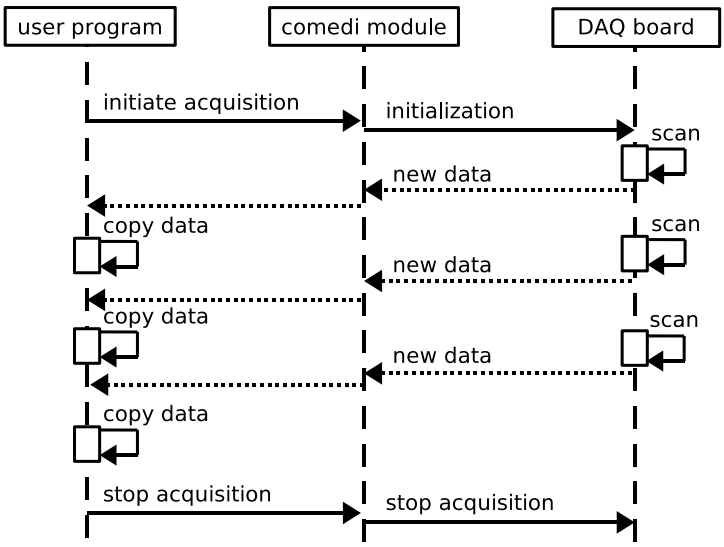


Figure 10.9.: Asynchronous Data Acquisition Sequence

At start time, the application program sends a DAQ requirements list (*scan* configuration) to the Comedi module. If the DAQ board's hardware is powerful enough (with respect to the overall sampling frequency, reading from multiple channels, etc.) and the periodic acquisition tasks are schedulable, the scan job is admitted and the data acquisition starts. From that time on, the application program is responsible for continuously fetching the sensor data from a previously allocated shared memory area. It has to be ensured that the application program is fast enough. Otherwise, unread input data will be overwritten. Generally, the application program is informed by semaphores when an appropriate portion of input data is available. Thus, the overall data acquisition concept

can be characterized as *push-based* and is therefore well-suited for the use within a data stream management system.

10.3.3. QStream Data Acquisition Strategies

For asynchronous data acquisition, a scan configuration has to include all parallel DAQ tasks. Figure 10.10 depicts a scan configuration, where a scan contains a number of n consecutive sensor readings.

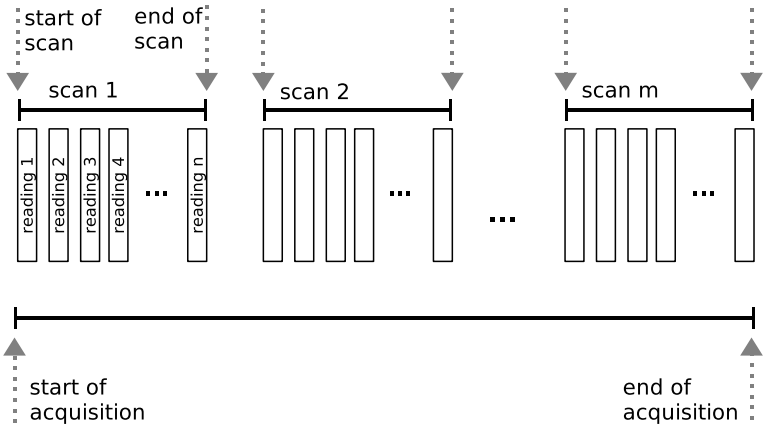


Figure 10.10.: Scan Configuration

Whether or not a scan can be put into effect depends on the DAQ board capabilities. First, the number of sensor readings within a scan is limited. Second, the minimum distance between two consecutive readings cannot be chosen arbitrarily small, as the DAQ board comes along with an upper bound regarding the overall sampling frequency. In addition, the DAQ board must provide the number of channels the DSMS requires, and the input voltage ranges (sensitivity) must meet the sensors' output characteristics.

Once admitted, a scan is executed repeatedly and forms the basis for continuously reading data from the desired sensors. From the viewpoint of a DSMS, the challenge now is to merge all DAQ requirements from one or more standing queries into one scan configuration.

Therefore, first, a list of source acquisition requirements based on the standing query configurations is created. This includes the source input channel which is to be scanned and the sampling frequency which shall be used. From additional information sources, the input sensitivity (voltage range) must also be given as parameter. Then, a *Comedi command* representing the scan is created out of the requirements. The Comedi command is tested against the DAQ hardware's capabilities and–if it is admitted–the DAQ process can be initialized and started. Finally–and during the lifetime of the appropriate standing queries–the data are fetched from the sensors and provided to the DSMS operator instances as input streams.

10.4. Summary

The current QStream implementation status includes a centralized DSMS with a query engine, a controller, a statmon and a monitor GUI component. A couple of standing queries are pre-configured for proving and testing the scheduling and adaptation strategies. The queries consist of elementary operators like filter, sampling, aggregation and join. Furthermore, comprehensive monitoring facilities are implemented, which allow for observing data stream characteristics and provide low-level information about operator processes and FIFO queues.

The sensor data acquisition concept is partially realized: In synchronous mode, sensor data can be used as input for standing queries, whereas in asynchronous mode, the DAQ process is implemented as an independent RTAI hard real-time component.

As future work for implementation, both (synchronous and asynchronous) DAQ modes should be made completely available for standing query evaluation. Furthermore, the output facilities of the DAQ boards should be addressed to provide control information for peripheral devices. In addition, it would be interesting to distribute the query engine across different nodes and make use of real-time-capable network drivers for implementing the distributed data flow. Regarding the real-time environment, QStream could be taken to other (RT)OS environments and the QoS requirements could be weakened in terms of tolerating a certain amount of exceeded deadlines (soft real-time). Last but not least, a flexible and comfortable graphical user interface for specifying standing queries is required. Alternatively, a declarative query language like CQL could be adapted and implemented.

11. Evaluation

The evaluation section's goal is to complement this thesis by presenting experiments which corroborate the resource calculation approach and the scheduling strategies of Chapter 8 as well as the robustness considerations of Chapter 9. First, Section 11.1 describes the hardware and software environment which was used for the evaluation of the QStream prototype. Then, Section 11.2 presents statistics, or rather data stream characteristics (DSCs), from QStream's filter which stand as examples for all implemented operator instances. Thereafter, Section 11.3 discusses the scalability of the data rate and runtime scheduling strategies simply based on the relationship of two consecutive operators. Section 11.4 completes the former scalability experiments by calculating the resources required by an example standing query. Finally, Section 11.5 corroborates the robustness concept by testing adaptation strategies and determining the robustness curve of a given configuration.

11.1. Test Environment Setup

All experiments have been conducted locally on an Intel Celeron PC with a CPU frequency of 2.8 GHz and 512 MB amount of main memory. The operating system environment consists of a Linux with kernel version 2.6.8.1, an Adeos 2.6r7/x86 patch and RTAI version 3.1. The periodic scheduling mode is used and the timer is initialized with a period length of $20\mu s$ ($20.114ns$) The timer's period length may also be chosen smaller to achieve higher accuracy, but–as a drawback–the overhead of context switching would not be tolerable any longer. A value of $20\mu s$ was found to be a good compromise.

All operator instance time measurements have been performed using the `rt_get_exec_time (...)` RTAI function, which reads out the CPU cycles which a real-time process (operator instance) has used. Then, this value is scaled with the duration of such a CPU cycle, which in turn depends on the server's processing speed and is $2,802,385$ throughout the experiments.

The operator's output sizes have been determined by simply counting the outgoing tuples. Furthermore, for data rates, the abbreviation *tps* (tuples per second) is used.

11.2. Scheduling Parameter Determination

Within the resource calculation chapter, the processing time t as well as the output size bo of an operator instance were assumed to be available from the DSC repository.

11.2.1. Operator Instance Processing Times

Figure 11.1 depicts the processing times of the individual filter operator's runs. Curves of four different microperiod values ranging from 1 to 250 are given. Obviously, the processing time increases as more microperiods are executed. Additionally, the absolute time jitter (as distance from the smallest to the largest time value within an individual curve) also increases.

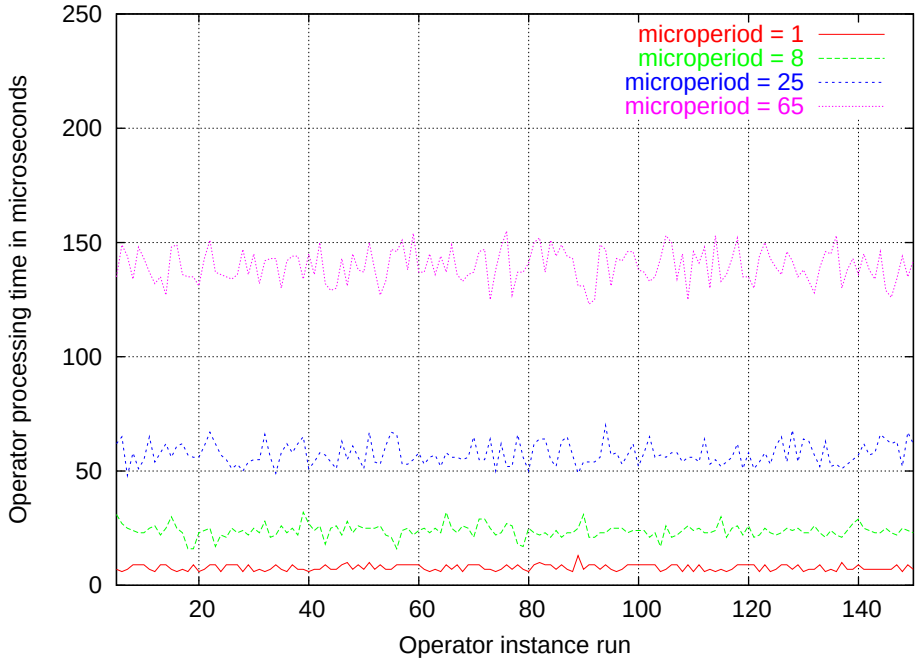


Figure 11.1.: Influence of Microperiods on Operator Instances' Processing Times

For a better illustration, Figure 11.2 shows the processing times per operator run by simply dividing the measured processing time values by the appropriate number of microperiods. It can be seen that the per-run processing time rapidly decreases as the number of microperiods grows. In other words, if each operator run were to be scheduled individually ($MP = 1$), the result would be a huge scheduling overhead.

Going one step further, Figure 11.3 allows for reasoning about the processing time jitter depending on the selected number of microperiods. It shows the processing time jitter (absolute and cumulated maximum value, absolute minimum value) scaled to the appropriate per-run processing time. This leads to the conclusion that–for a low number of microperiods and thus for only short units of work with small measured times–the processing time jitter values are extraordinarily high and thus should not be considered for resource reservation. From a value of $MP \geq 100$ on ($t \geq 70\mu s$ in case of the filter operator instance), the relative time jitter becomes moderately low and not too many resources are wasted when these times are used as scheduling basis.

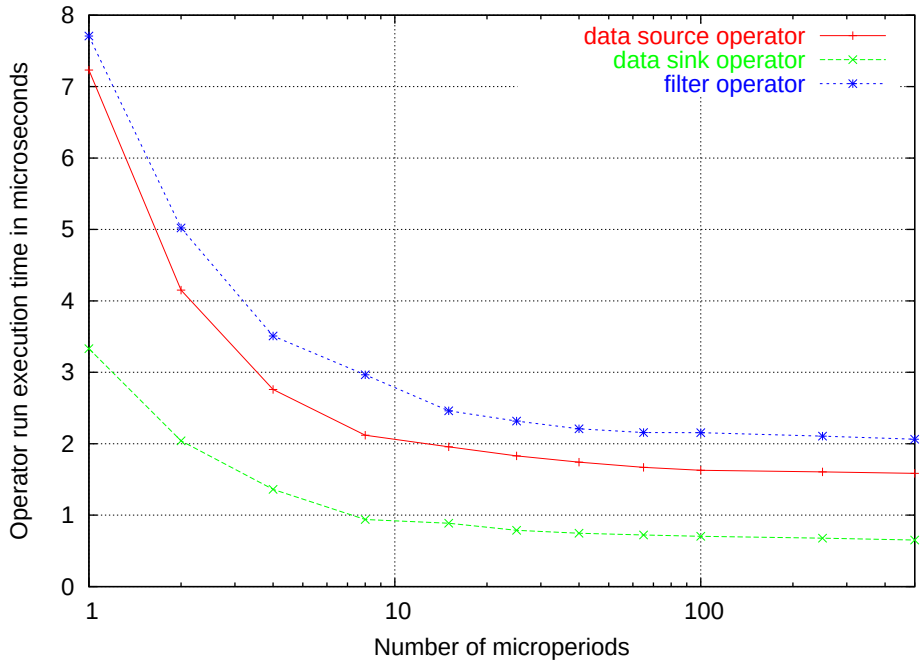


Figure 11.2.: Influence of Microperiods on Scheduling Overhead

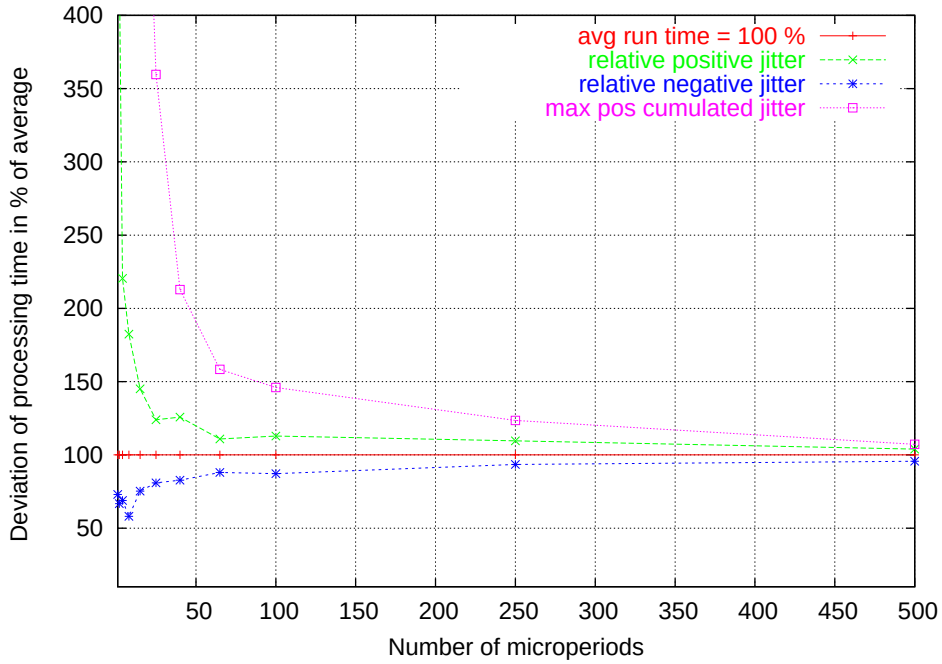


Figure 11.3.: Influence of Microperiods on Relative Time Jitter

11.2.2. Operator Instance Output Volume

Now, the data volume output characteristic including output jitter is illustrated as example for the filter operator in the diagram in Figure 11.4. The filter has a selectivity of 0.4969, which equals the average output size *bo* per run. The output jitter depends on the input data attribute values: if the attribute value fulfills the filter predicate, it is passed by and the curve increases by one element during a run. Otherwise, if the attribute value is discarded, the output size remains at the previous level.

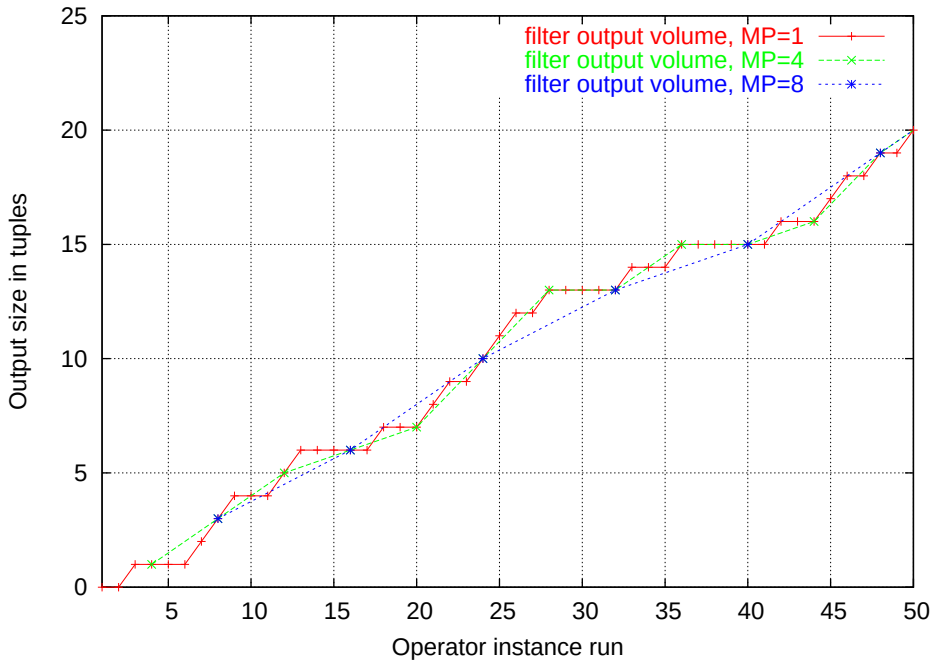


Figure 11.4.: Filter Output Volume Example

Within the diagram, the output behavior of three different numbers of microperiods is given. As one can see, the output batch jitter becomes smaller as the number of microperiod grows (the blue curve of $MP = 8$ has a much smoother appearance than the curves of $MP = 1$ or $MP = 4$). This is obvious because the filter output is 'balanced' more and more when considering a number of consecutive operator instance runs.

Figure 11.5 depicts the filter operator's cumulated output jitter (input data configuration and filter predicate remain the same) to illustrate the output jitter throughout a large time span. The cumulated minimum $\sigma^{\ominus}$ and maximum $\sigma^{\oplus}$ are annotated appropriately. Although the input data was supposed to be equally distributed and the filter's selectivity was to be 0.5, a cumulated minimum jitter of $\sigma^{\ominus} = 43$ and a cumulated maximum jitter of $\sigma^{\oplus} = 25$ arise during runtime and must therefore be considered as input parameter for the JCP+ resource calculation.

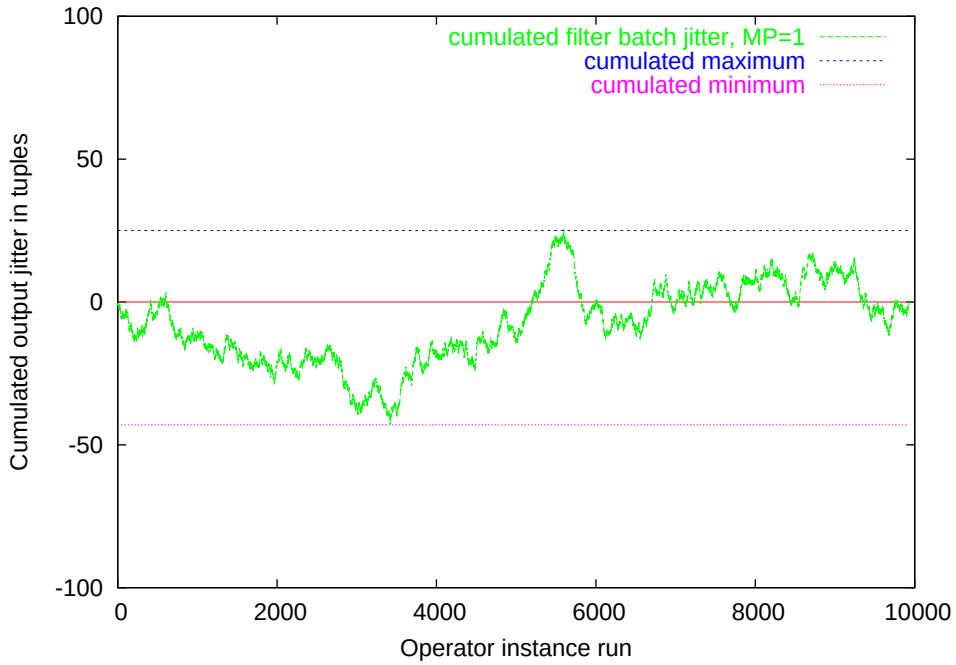


Figure 11.5.: Cumulated Filter Operator Output Jitter

11.3. Scalability of Scheduling Strategies

This section reviews the individual scheduling strategies and contemplates them from the perspective of scalability. Therefore, Table 11.1 first names the different scheduling configurations. It is equivalent to Table 8.1 from Section 8.3.1.

	Run Time Scheduling	
	Min Delay (MD)	Max Throughput (MT)
Data Rate Scheduling		
Avg Data Rate (ADR)	configuration I	configuration III
Max Data Rate (MDR)	configuration II	configuration IV

Table 11.1.: Combination of Scheduling Strategies

On that basis, the scheduling strategies are compared pairwise. First, Section 11.3.1 discusses the resource consumption of the MD and MT runtime scheduling strategy (Configuration I versus Configuration III). Then, in Section 11.3.2, the jitter influence on the query resources for the ADR and MDR data rate scheduling strategy are compared (Configuration I versus Configuration II).

11.3.1. Run Time Scheduling Strategy Comparison - Configuration I versus Configuration III

The first part of this section examines the influence of an operator's processing time jitter ($\tau^{\perp}$ and $\tau^{\top}$) on the required CPU utilization. Four curves of CPU utilization depending on the data rate are displayed in Figure 11.6. One belongs to the Max Throughput strategy, the other three represent the Min Delay strategy with different values of $\tau^{\perp}$ and $\tau^{\top}$ (where $\tau^{\perp}$ equals $\tau^{\top}$).

Two conclusions can be drawn, both of which approve the scheduling strategies' motivation: First, the CPU utilization of both runtime scheduling strategies increases linearly with the data rate at which the standing query is evaluated. Second, when using the Max Throughput strategy, the processing time jitter of an operator instance does not have any effect on the CPU utilization because it is only the average run time of an operator instance that was considered for the determination of the CPU utilization. For that reason, additional MT graphs are left out in the diagram. The CPU utilization of the Min Delay curves increases (linearly) with the amount of time jitter because $\tau^{\perp}$ and $\tau^{\top}$ must be added completely to the average runtime when determining the CPU utilization (worst-case consideration).

The drawbacks of the MT strategy are the increased inter-operator delays and, as a result, the higher buffer requirements. Figure 11.7 puts the focus on this issue and compares the inter-operator delay of both strategies depending on the time jitter of an operator instance (which is to be interpreted as $\tau^{\top}$ in MD strategy and as $\tau^{\oplus}$ in MT strategy). Both the red and the green curve increase linearly but the MT delay is significantly higher. The initial increase of the MT delay is caused by the consideration of

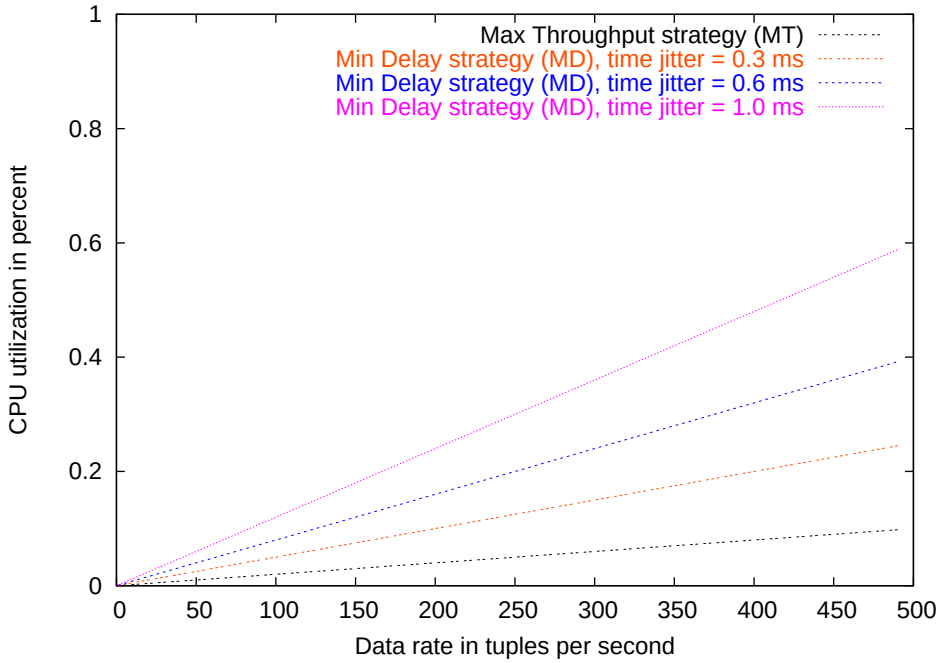


Figure 11.6.: CPU Utilization of MD versus MT

whole-numbered multiples of the operator instance's period length during the calculation. In addition to the delay graphs, the CPU utilization of MD and MT is shown as a dotted line.

If the CPU is working to its full capacity, no higher amounts of jitter can be considered in the MD strategy. Therefore (for sake of completeness), the scalability of buffer size and operator delay depending on larger jitter values $\tau^{\oplus}$ is given in Figure 11.8 exclusively for the MT strategy. Both the buffer resource and the delay QoS measure scale linearly with $\tau^{\oplus}$. Again, the 'staircase' behavior is caused by the consideration of whole multiples of the operator's period lengths when calculating the required resources. The CPU utilization within the experiment is 0.4 and remains constant.

11.3.2. Data Rate Scheduling Strategy Comparison - Configuration I versus Configuration II

In comparison to the former set of experiments, now the Avg Data Rate strategy is compared with the Max Data Rate strategy. The diagrams in Figure 11.9 and 11.10 depict the intermediate buffer size requirements depending on the time jitter ($\tau^{\perp} = \tau^{\top}$) of the producer operator.

Both batch and time jitter have linear influence on the intermediate buffer size. For MDR, the memory resources are lower than for ADR, as buffer underruns are tolerated and thus only the jitter $\tau^{\top}$ must be considered. Due to the fact that $\tau^{\perp}$ equals $\tau^{\top}$, the MDR buffer requirements are exactly half of the ADR buffer requirements. The cause

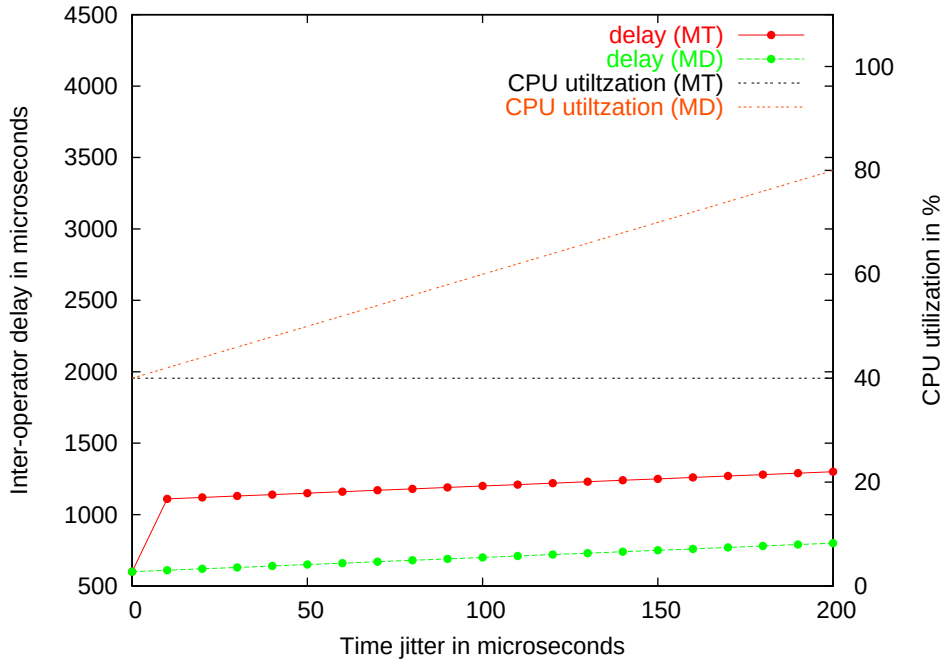


Figure 11.7.: Scalability of Output Delay: MD versus MT (CPU Utilization of MT is Constant at 0.4)

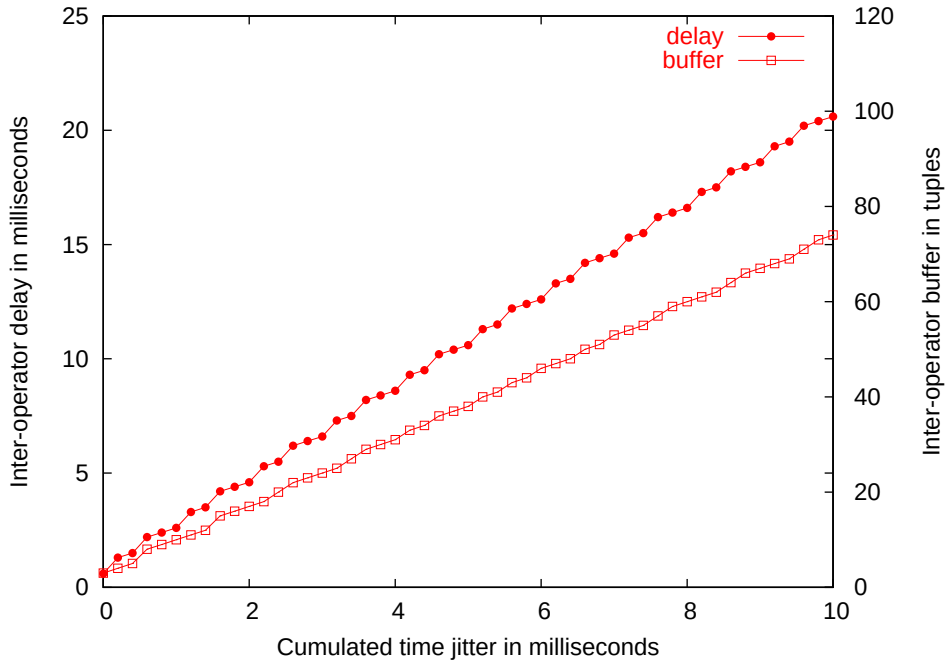


Figure 11.8.: Scalability of Output Delay and Buffer Size in MT

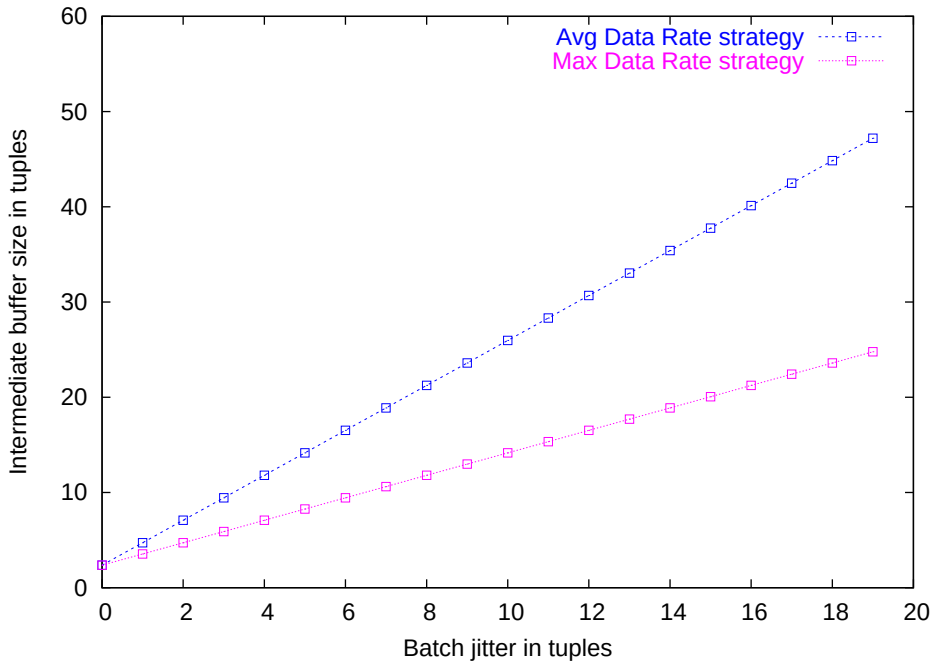


Figure 11.9.: Scalability of Buffer Size: MDR versus ADR

for the 'staircase' effect in the diagram in Figure 11.10 is the rounding-off of the buffer size to the next whole-numbered amount of tuples.

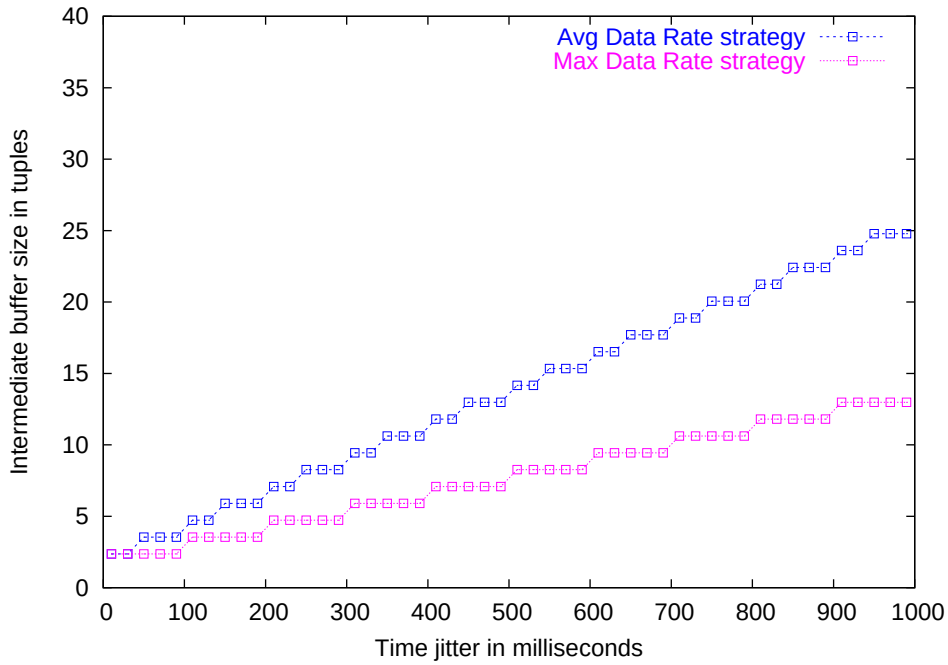


Figure 11.10.: Scalability of Buffer Size: MDR versus ADR.

11.4. Example Query Resource Consumption

The focus of the experiments is now shifted to a larger example query which is given by Figure 11.11. Accordingly, Table 11.2 states the deployed operators along with the query-specific parameter settings.

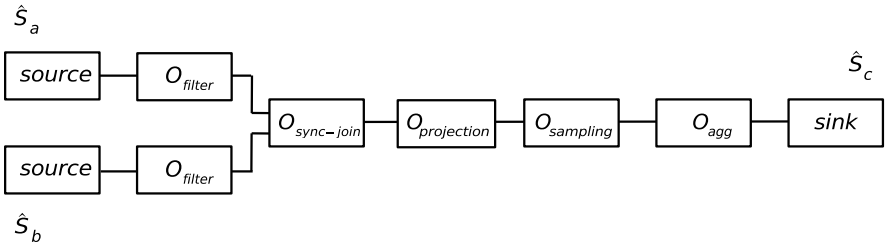


Figure 11.11.: Example standing query instance QI

11.4.1. Description of Operators and Operator Instances

Operator	Parameters	Description
source		delivers randomly distributed numerical data (two-tuples, event stream ES)
O_{filter}		filter predicate on second attribute
$O_{sync-join}$	$mode = OUTER$	
$O_{projection}$		passes by timestamp and second attribute value
$O_{sampling}$	$N = 6$, $n = 3$	stratified sampling
O_{agg}	$f_{agg} = AVG$, $f = 10$	jumping window aggregation
sink		reads out the tuples from the last FIFO queue

Table 11.2.: Description of elementary operators

The operators of the example query are listed in Table 11.2 and the corresponding operator instance descriptions are given in Table 11.3. There, each operator instance's period length and input batch size are listed as an excerpt of data rate requirements the query runs on. Due to the large absolute time jitters at a lower number of microperiods, a value of $MP = 100$ was selected for this experiment. This means that within each scheduled period, an operator instance is executed exactly 100 times.

Table 11.4 lists the DSCs of the operator instances. They are specific for the test server as well as for the example standing query. The given time as well as output size statistics (except *bo*) refer to the configuration of $MP = 100$, too. In detail, there are the average runtime per period t, the absolute and cumulative time jitter values $\tau^{\perp}$, $\tau^{\top}$ and $\tau^{\oplus}$, the average number of output tuples per run *bo*, and the cumulative size jitter $\sigma^{\oplus}$ and $\sigma^{\ominus}$, respectively.

Operator	**Period Length** (μs) **for a given Result Data Rate** (tps)						**Input Batch Size** ($tuples$)
	100	**200**	**...**	**1000**	**...**	**2000**	bi
$source$	24988.00	12494.00		2498.80		1249.40	—
OI_{filter}	24988.00	12494.00		2498.80		1249.40	1
$OI_{sync-join}$	50000.00	25000.00		5000.00		2500.00	1
$OI_{projection}$	50000.00	25000.00		5000.00		2500.00	1
$OI_{sampling}$	300000.00	150000.00		30000.00		15000.00	6
OI_{agg}	100000.00	50000.00		10000.00		5000.00	1
$sink$	1000000.00	500000.00		100000.00		50000.00	1

Table 11.3.: Operator Parameters

Operator	**Times Characteristics**(μs)				**Size Characteristics** ($tuples$)		
	t	$\tau^{\perp}$	$\tau^{\top}$	$\tau^{\oplus}$	bo	$\sigma^{\ominus}$	$\sigma^{\oplus}$
$source$	189.2	25.76	4.24	328.9	1	0	0
OI_{filter}	211.5	32.46	24.54	356.2	0.49976	152.2	93.51
$OI_{sync-join}$	320.3	25.66	8.338	695.1	1	0	0
$OI_{projection}$	276	29.03	10.97	527.6	1	0	0
$OI_{sampling}$	806.5	16.47	7.531	155	3	0	0
OI_{agg}	172.3	22.66	8.336	346.1	0.1	4	8
$sink$	168.1	5.913	4.087	14.74	0	0	0

Table 11.4.: DSCs of Example Operator Instances

The time jitter characteristics of all participating operator instances are visualized in Figure 11.12 in excerpts. The filter operator instance jitters most, followed by the aggregation operator instance. Both of them also incorporate size jitter when producing data. The filter operator instance's output depends on whether the stream tuples fulfill the filter predicate. The number of tuples which belong to one (time-based) aggregation group is determined by the tuples' internal timestamp information and may vary, too. The cumulative output size jitter of OI_{filter} and OI_{agg} is illustrated in Figure 11.13. The appropriate minimum and maximum values of Table 11.4 can be read directly from the diagrams.

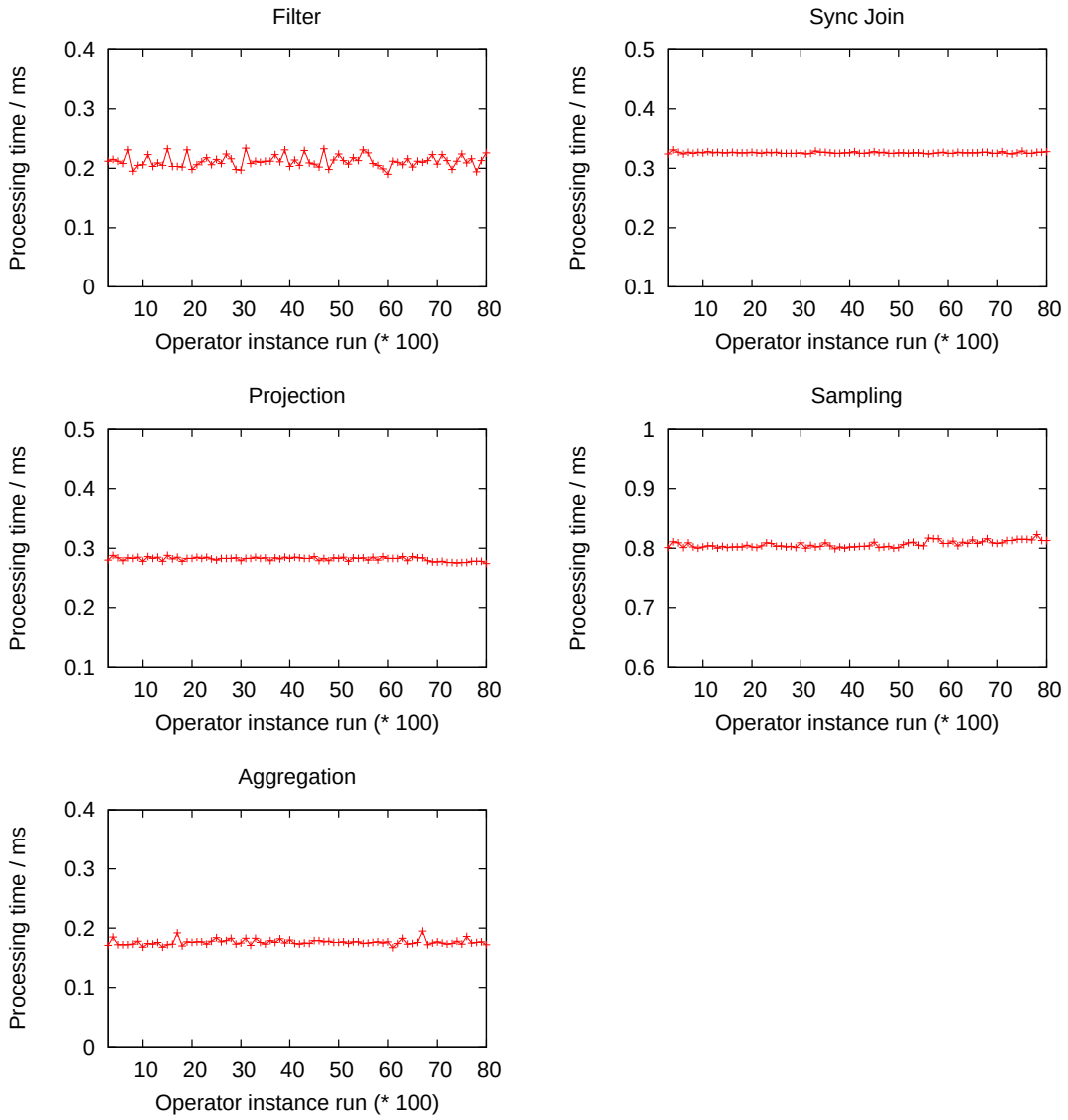


Figure 11.12.: Processing Time Characteristics of Example Operator Instances (MP=100)

11.4.2. Example Query Resources and Quality-of-Service

Finally, query QI gets scheduled with Average Data Rate strategy. There, the resource consumption in terms of intermediate buffer sizes and CPU utilization, together with the result QoS in terms of output delay, is given for different result data rate requirements R_{min} (Table 11.5). The basis is the appropriate formulas of the Min Delay and Max Throughput runtime scheduling strategy. Therefore, the operator instance's batch sizes (input and output) have been multiplied with the selected number of microperiods to

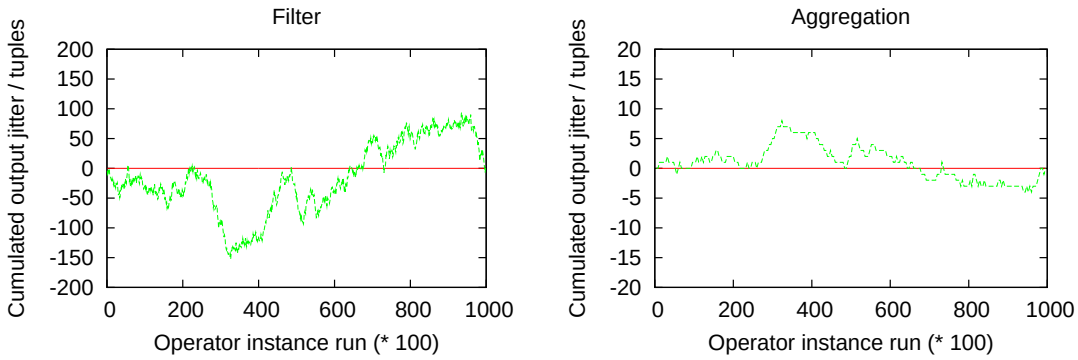


Figure 11.13.: Batch Size Characteristics of Example Operator Instances (MP=100)

get the correct result (i.e. if $bo = 3$ and $MP = 100$, a value of $bo' = 300$ must be used instead).

The output delays for the application of different scheduling strategies to QI are illustrated in the diagrams in Figure 11.14 and 11.15. First, in Figure 11.14, the output delays of the MT and MD strategies are compared, both depending on the required result data rate R_{min}. Note that the CPU utilization graphs stop at a value of 69% because with higher utilization, the scheduling is not QoS-guaranteeing any longer, as not all operator instance period deadlines can be met (overload criteria of RMS scheduling). This implies that a maximum data rate of 1200 *tps* (MD) and 1400 *tps* (MT) can be achieved. If higher data rates are used, the data exchange may become blocking.

It can be seen from the delay curves that the higher the data rate (and thus the higher the CPU utilization), the larger the output delay of the MT strategy becomes. The reason lies in the jitter compensation process. It depends on the remaining CPU time, which decreases continuously at higher data rates. In contrast, the output delay of MD seems to remain constant.

Figure 11.15 focuses on the data rate range where the CPU utilization is below the limit of 69% and thus allows the observation of the behavior of the output delay in more detail: First, the output delay of both runtime scheduling strategies decreases with a growing data rate because–if the standing query instance works faster–the tuples are handed over from operator to operator more quickly. Then, at a data rate of about 1000 *tps*, the remaining CPU time becomes too small to compensate all the time jitter of the operators within the current operator instance's period. As a result, the consecutive operator instances are delayed more and more to give them enough time to compensate their time jitter. This directly influences the output delay and–although not shown in the diagrams–the FIFO queue sizes.

The data rate from which on the output delay of MT starts to increase depends on the amount of cumulated operator instance time jitter: If more jitter has to be covered, the output delay of MT will increase at lower data rates due to the limited amount of remaining CPU time.

Result Data Rate Requirement (*tps*)	**Result Resources and QoS**					
	MD Strategy			**MT Strategy**		
	CPU Util.	**FIFO Mem** (*tuples*)	**Output Delay** (*ms*)	**CPU Util.**	**FIFO Mem** (*tuples*)	**Output Delay** (*ms*)
100	0.0544106	2709	1644	0.04849	3317	1744
200	0.108821	2717	823	0.0969801	3325	873
300	0.163232	2723	549	0.14547	3331	582
400	0.217642	2730	412	0.19396	3338	437
500	0.272053	2739	330	0.24245	3347	350
600	0.326463	2745	275	0.29094	3353	292
700	0.380874	2752	236	0.33943	3360	250
800	0.435284	2758	207	0.38792	3366	219
900	0.489695	2764	184	0.43641	3372	195
1000	0.544106	2773	166	0.4849	3381	176
1100	0.598516	2779	151	0.53339	13337	815
1200	0.652927	2786	138	0.58188	60534	3530
1300	(0.707337)	2794	128	0.63037	84552	4651
1400	(0.761748)	2800	119	0.67886	101718	5262
1500	(0.816158)	2810	111	(0.727351)	146278	7178
1600	(0.870569)	2816	104	(0.775841)	405104	18973
1700	(0.92498)	2823	98	(0.824331)	637360	28504
1800	(0.97939)	2829	93	(0.872821)	1110017	47476
1900	(1.0338)	2836	88	(0.921311)	2055104	83693
2000	(1.08821)	2845	84	(0.969801)	5069213	189344

Table 11.5.: Resources and QoS of Example Query

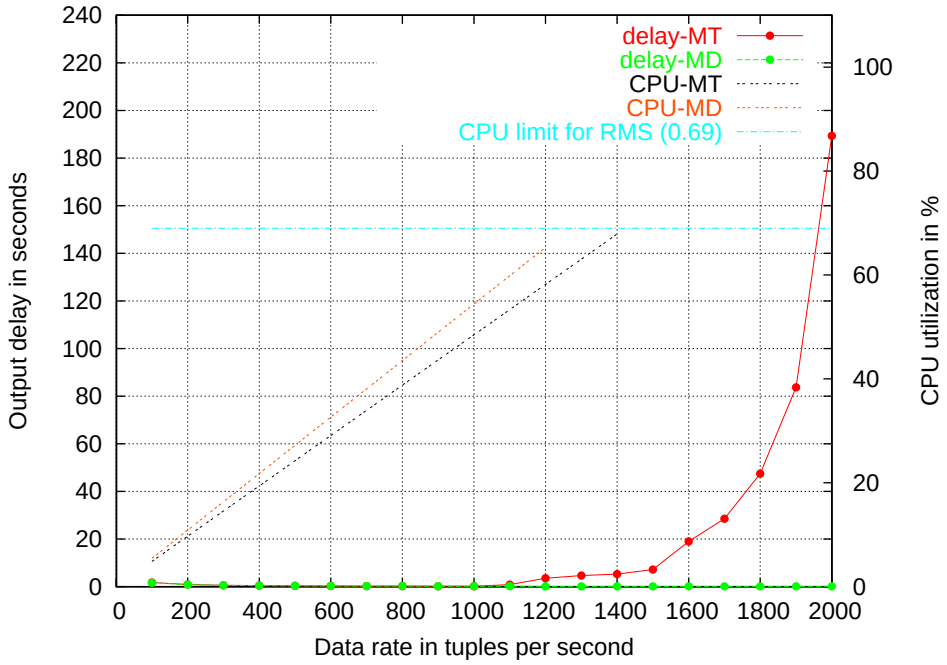


Figure 11.14.: Output Delay of MT Strategy Depending on Data Rate

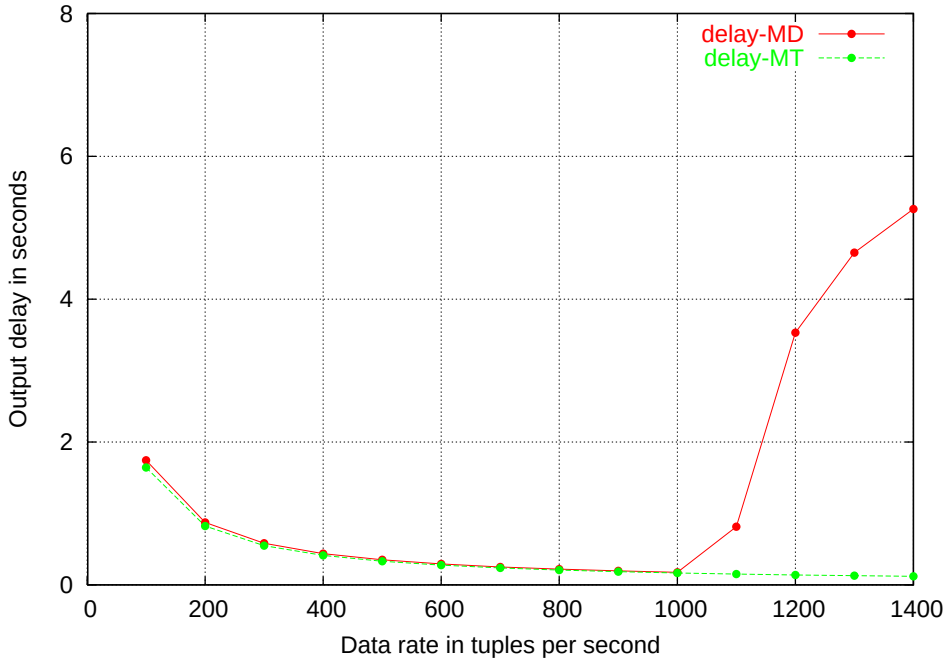


Figure 11.15.: Output Delay of MT Strategy Depending on Data Rate - Schedulable Range

11.5. Adaptation and Robustness

The DSMS' ability to adapt to new environmental situations has been examined using a simple test standing query consisting of three operators (Figure 11.16).

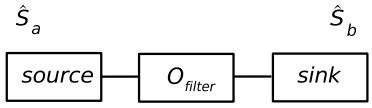


Figure 11.16.: Example Standing Query Instance for Adaptation Experiments

The data source delivers a stream containing 100,000 random attribute values ranging from 0 to 99. Thereafter, a filter operator passes by only those tuples that have an attribute value smaller than 50. The analysis of the source data gives a filter predicate selectivity of exactly 0.49976. Furthermore, the filter output stream contains a large cumulated batch jitter of $\sigma^{\ominus} = 164$ *tuples* and $\sigma^{\oplus} = 104$ *tuples* (Figure 11.17), which has to be compensated by the consecutive FIFO buffer. The number of microperiods is 1 throughout the adaptation experiment. The required result data rate is $R_{min} = 500$ *tps*.

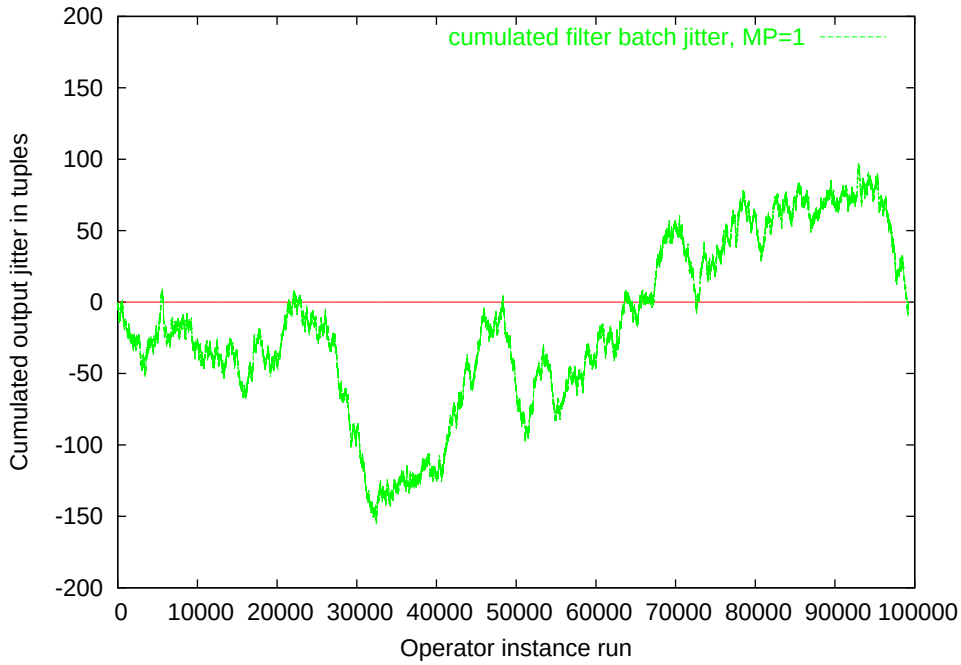


Figure 11.17.: Cumulated Filter Batch Jitter ($MP = 1$)

The goal of this experiment is to show that the smaller the granted resources in terms of filter operator output buffer size, the more adaptations take place, and thus, the lower the DSMS robustness. Therefore, the appropriate FIFO queue size is decreased step by step, starting with a sufficient size of 268 *tuples*. The DSMS is monitored performing

adaptations and the robustness value Ω is calculated from the number of adaptations and the monitor time span. Two aspects of adaptation are illustrated in Figure 11.18. First, the diagram depicts the adaptation time points of the individual experiments.

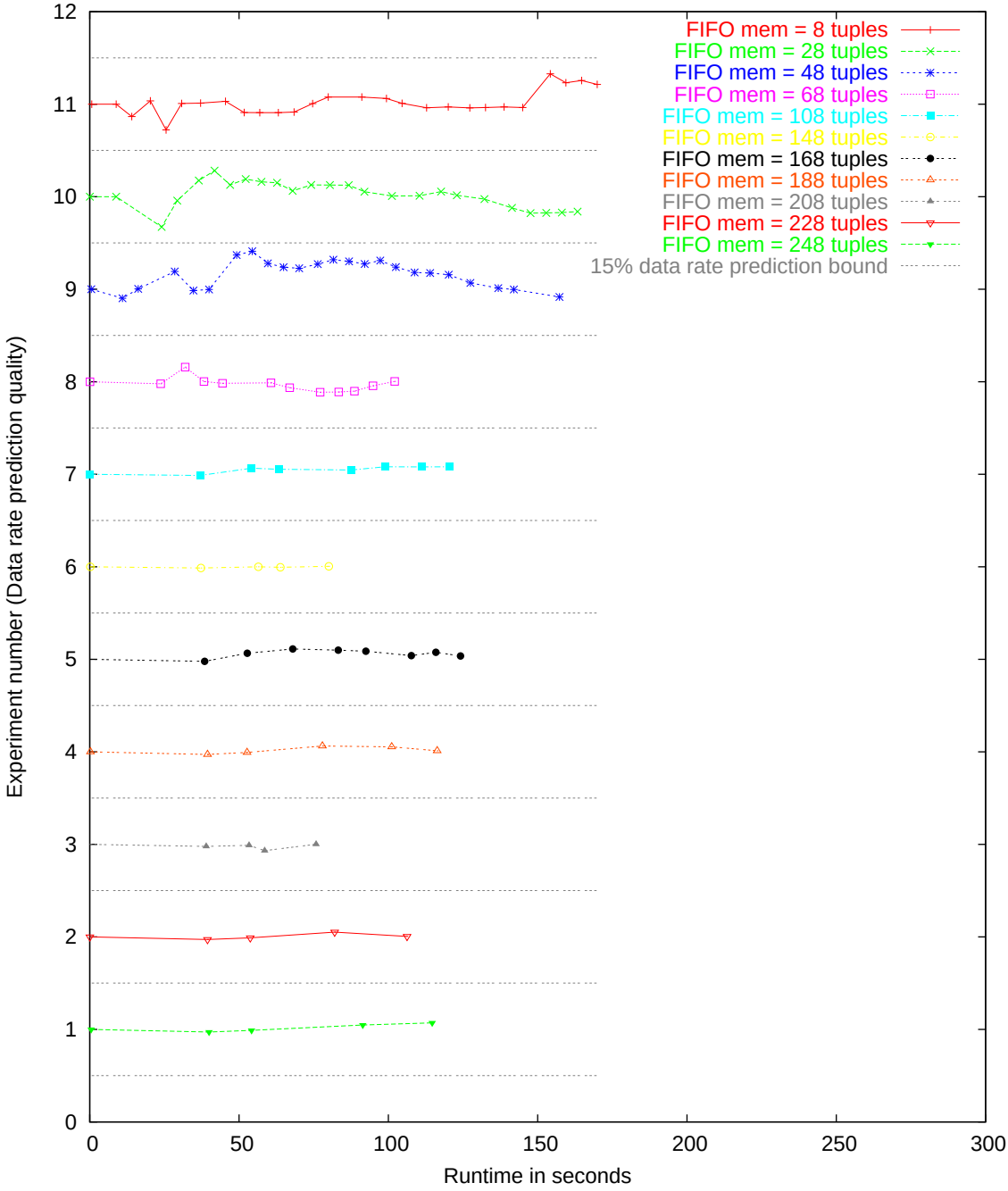


Figure 11.18.: Adaptation Time Points and Predicted Filter Output Data Rate

The graph with sufficient FIFO queue memory (268*tuples*) is left out here because no adaptation occurs during the whole runtime. It becomes clear that the number of adaptations increases as more FIFO memory is granted. The duration between the query evaluation start and the first adaptation decreases monotonically, whereas the overall number of adaptations increases in general but not monotonically (as the graphs of test runs 5 and 6 show).

Furthermore, the adaptation time points are extended by the value of the predicted filter output data rate. At starting time, the data rate is 500*tps* in each test, run which acts as the basis for the graph. During runtime, each participating adaptation changes the data rate, which is qualitatively shown as a percentage of the initial value. For data rate prediction, the simple average calculation based on historical values is used. When doing so, one must know how the adaptation procedure of the prototype works: For the initial standing query configuration, the result data rate requirement $R_{min} = 500$ *tps* is propagated upwards to obtain the data rates as well as the period lengths of all participating operator instances. Then, if an adaptation is triggered, the newly predicted data rates are (only) propagated down to the data source (opposite direction). As a result, the filter output, and thus the data sink input data rate (which were initially set to a value of exactly 500 *tps*), is changed repeatedly. If adaptations take place, the new currently predicted data rate is propagated down to the data sink, whereas the data rates of the operators upstream to the adapted one (in this case the filter and the data source) remain constant. For that reason, only the filter's output data rate or the data sink's input data rate vary in the experiment.

It can be seen that the number of adaptations increase as the granted resources (FIFO memory) are reduced. Furthermore, the data rate prediction gets worse with lower memory resources. This in turn can lead to deviations of about 15 percent between the real average value and the predicted one. The reason is that–if adaptations take place very frequently–the history of DSC values stored in the DSC repository is interrupted, and thus, only the (very few) most recent values (which have arrived *after* the last adaptation) can be considered when predicting the new filter output data rate.

Finally, the robustness value Ω is calculated from the individual experimental results. One obtains a robustness curve as illustrated in Figure 11.19.

Two things need to be mentioned here: First, the more FIFO memory is granted for a given standing query evaluation, the fewer adaptations take place and the higher the robustness value Ω. Second, the robustness values does not necessarily increase monotonically with the amount of FIFO memory. As the test runs 6 (148 *tuples* buffer size) and 7 (108 *tuples* buffer size) show, the robustness may also decrease temporary depending on the batch jitter distribution over time and depending on the predicted new data rate.

11.6. Summary

The evaluation showed that the resource calculation model as well as the scheduling strategies of QStream meet the requirements of QoS-aware data stream processing. It

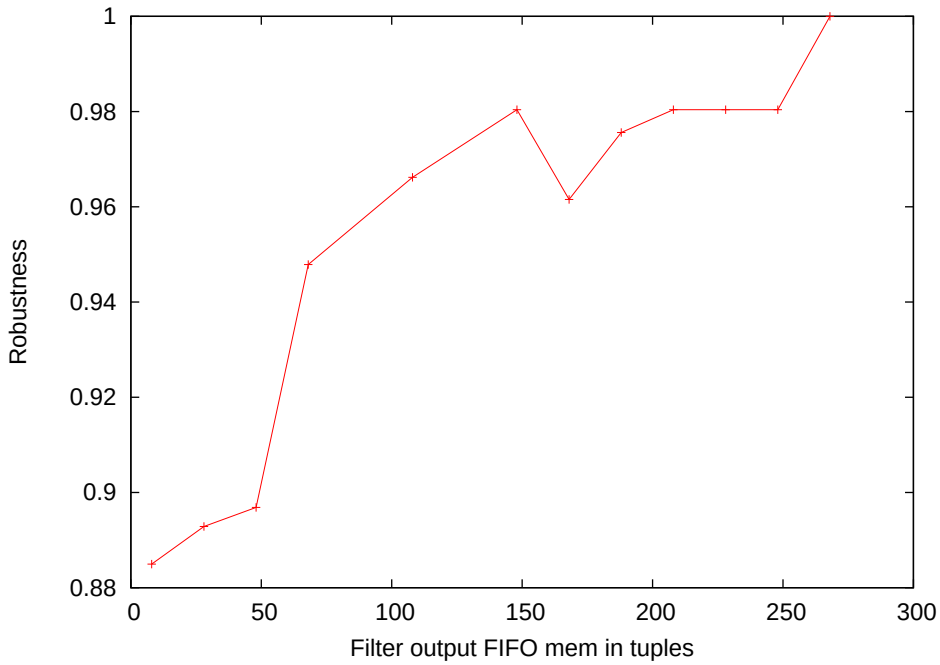


Figure 11.19.: Robustness Curve

was confirmed that–if resources for covering the worst case of data stream characteristics are available–the negotiated QoS can be provided during the lifetime of the standing query. Moreover, the resource allocation is adapted to the query evaluation process to not waste any memory and to not unnecessarily delay the result data. If occasional interruptions of the QoS guarantees can be tolerated, the robustness concept may be applied. The trade-off between the amount of query evaluation resources and the time span during which a query runs without any adaptation was impressively demonstrated.

Part IV.

Summary

QoS-Aware Data Stream Processing

Data stream processing in a QoS-aware manner is a challenging area in current research and development activities. New applications in the field of data management require the efficient processing of transient data which are disseminated in the form of streams. In this context, existing concepts and implementations are first exploited and adapted to the processing paradigms of transient data and permanent (standing) queries. Second, data stream processing is often connected with sensor data management, which comes along with a variety of hardware-oriented solutions, mostly implemented on digital signal processing hardware. The challenge at this point is to keep up with the predictability and–more specifically–with the real-time capability of such solutions. This thesis put the focus on this very issue by providing Quality-of-Service guarantees for the DSMS and paying particular attention to the evaluation of standing queries with associated time constraints. Moreover, the flexibility of a pure software-based solution is combined with the predictability of a hardware-oriented one.

Modeling Aspects

The thesis started with a comprehensive overview of related work. Thereafter, it presented a novel data stream processing model which allows for stream classification based on the temporal data source behavior and distinguishes different data stream classes. Furthermore, content-based as well as time-based QoS metrics belong to each data stream. The content-based QoS metrics (signal frequency and inconsistency) describe how the original sensor data are reflected within the data stream. In a broader sense, the signal frequency is associated with the stream's information content, whereas the inconsistency stands for an error measure regarding the relation of timestamps and attribute values. The thesis' focus was clearly on the time-based QoS metrics (data rate and delay). They describe the temporal aspect of data stream processing by annotating the data rate as well as a delay property for each intermediate stream and by propagating them from the data sources to the data sink. The challenge of providing QoS guarantees is directly associated with meeting the overall requirement for the delay and the result data rate at the output stream of the last operator. Therefore, dedicated resource calculation and scheduling strategies are given.

Cost Calculation and Scheduling Strategies

The thesis presented the JCP+ cost calculation approach. It allows for calculating operator- as well as stream-based resources required for evaluating a standing query with the demanded (time-based) result quality. Moreover, JCP+ results in the overall delay based on a given standing query evaluation speed.

For providing QoS guarantees, it is essential to efficiently map the calculated resources to the operating system underneath and thus to make an appropriate reservation. In this regard, the thesis presented scheduling strategies for considering alternative operator processing times and alternative data rates as reservation basis. This is beneficial as it allows to trade off required resources like memory and CPU consumption. Moreover, it

is also possible to trade resources (e.g. CPU consumption) against quality metrics (e.g. overall delay) and thus to have some clearance during the QoS negotiation process.

Robustness Concept

The resource calculation basis is the characteristics of the data streams and of the operators which are initially estimated and then continuously improved by monitoring the query evaluation. In this context, the overall goal is not to find 'perfect' overall charactaristics. As data streams change over time, it is more important to quickly detect such changes and to obtain possible future characteristics using an adequate prediction model. The QStream robustness concept comprises trading the steady query evalation process against the amount of reserved resources. The more resources one spends on covering jitter in terms of operator processing time or output data volume, the longer the DSMS runs without an interruption of the continuous data flow, and the higher the system's robustness. The relationship between the amount of resources and robustness is described by the query-specific robustness curve.

Future Work

Providing hard limits regarding the Quality-of-Service requirements comes with the need for sufficient resources to cover the worst case, e.g. regarding the processing time jitter or regarding the output volume jitter. Weakening the hard Quality-of-Service constraints (soft QoS) could be promising if the surrounding application allows it. In a broader sense, the robustness concept acts as a first step in this direction; it may be extended by future work. Another interesting issue–although not discussed in this thesis–is distributed stream processing while keeping up (hard or soft) QoS guarantees. The processing concepts of QStream could either be distributed to a set of participating servers or directly pushed down to the sensor devices which are able to acquire and pre-process sensor data in the form of streams.

List of Figures

List of Tables

Glossary

ADR	Average Data Rate scheduling strategy
API	Application Programming Interface
CQL	Continous Query Language - Anfragesprache in STREAM
CS	Continuous Partial Stream
DAQ	Data Acquisition
DBMS	Data Base Management System
DS	Discontinuous Partial Stream
DSC	Data Stream Characteristics
DSMS	Data Stream Management System
ES	Event Partial Stream
FIFO	First-In-First-Out
GUI	Graphical User Interface
JCP	Jitter-constrained Periodic Streams
LXRT	RTAI library for providing soft and hard real-time in user space
MD	Min Delay scheduling strategy
MDR	Max Data Rate scheduling strategy
MQO	Multi-Query Optimization
MT	Max Throughput scheduling strategy
OS	Operating System
QEP	Query Execution Plan
QoS	Quality-of-Service
RMS	Rate Monitonic Scheduling
RTAI	Realtime Application Interface - Echtzeiterweiterung fr Linux
RTOS	Real-time Operating System
SHM	Shared Memory
SP	Scheduling Plan

Bibliography

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR 2005, January 4-7, Alisomar (CA), USA)*, pages 277–289, 2005.

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. STREAM: The Stanford Data Stream Management System. Technical report, Department of Computer Science, Stanford University, 2004.

[ABc+05] Yanif Ahmad, Bradley Berg, Ugur Çetintemel, Mark Humphrey, Jeong-Hyon Hwang, Anjali Jhingran, Anurag Maskey, Olga Papaemmanouil, Alex Rasin, Nesime Tatbul, Wenjuan Xing, Ying Xing, and Stanley B. Zdonik. Distributed operation in the Borealis stream processing engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD 2005, June 13-16, Baltimore (MD), USA)*, pages 882–884, 2005.

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *Database Programming Languages, 9th International Workshop (DBPL 2003, September 6-8, Potsdam, Germany)*, pages 1–19, 2003.

[ACc+03] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000, May 16-18, Dallas (TX), USA), 2000*, pages 261–272, 2000.

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing (STOC 1996, May 22-24, Philadelphia (PA), USA)*, pages 20–29, 1996.

[AN04] Ahmed Ayad and Jeffrey F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 2004, June 13-18, Paris, France)*, pages 419–430, 2004.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2002, June 3-5, Madison (WS), USA)*, pages 1–16, 2002.

[BBD+04] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, 2004.

[BBDM03] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwanis. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003, June 9-12, San Diego (CA), USA)*, pages 253–264, 2003.

[BDE+97] Claudio Bettini, Curtis E. Dyreson, William S. Evans, Richard T. Snodgrass, and Xiaoyang S. Wang. A Glossary of Time Granularity Concepts. In *Temporal Databases, Dagstuhl*, pages 406–413, 1997.

[BDM03] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding techniques for data stream systems. In *Proceedings of the 2003 Workshop on Management and Processing of Data Streams (MPDS'03, June 8, San Diego (CA), USA)*, 2003.

[BDM04] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load Shedding for Aggregation Queries over Data Streams. In *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004, March 30 - April 2, Boston (MA), USA)*, pages 350–361, 2004.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In *Proceedings of the Mobile Data Management, Second International Conference (MDM 2001, January 8-10, Hong Kong, China)*, pages 3–14, 2001.

[BMM+04] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive Ordering of Pipelined Stream Filters. In *Proceed-*

ings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 2004, June 13-18, Paris, France), pages 407–418, 2004.

[BSLH05] Henrike Berthold, Sven Schmidt, Wolfgang Lehner, and Claude-Joachim Hamann. Integrated Resource Management for Data Stream Systems. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing (SAC 2005, March 13-17, Santa Fe (NM), USA)*, pages 555–562, 2005.

[BSW04] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems (TODS)*, 29(3):545–580, 2004.

[CA04] Damianos Chatziantoniou and Achilleas Anagnostopoulos. NESTREAM: Querying Nested Streams. *SIGMOD Record*, 33(3):71–78, 2004.

[CcC+02] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002, August 20-23, Hong Kong, China)*, pages 215–226, 2002.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shahhah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR, January 5-8, Asilomar (CA), USA)*, 2003.

[CcR+03] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator Scheduling in a Data Stream Manager. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003, September 9-12, 2003, Berlin, Germany)*, pages 838–849, 2003.

[CD02] Jianjun Chen and David J. DeWitt. Dynamic Re-Grouping of Continuous Queries. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison (WI), USA, 2002.

[CDN01] Surajit Chaudhuri, Gautam Das, and Vivek R. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD 2001, May 21-23, Santa Barbara (CA) USA)*, 2001.

[CDN02] Jianjun Chen, David J. DeWitt, and Jeffrey F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proceedings of the 18th International Conference on Data Engineering (ICDE, February 26 - March 1, San Jose (CA), USA)*, pages 345–356, 2002.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000, May 16-18, Dallas (TX), USA)*, pages 379–390, 2000.

[CFPR00] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, and Anne Rogers. Hancock: A Language for Extracting Signatures from Data Streams. In *Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD 2000, August 20-23, Boston (MA), USA)*, pages 9–17. ACM, 2000.

[CHK+03] Michael Cammert, Christoph Heinz, Jürgen Krämer, Alexander Markowetz, and Bernhard Seeger. PIPES: A Multi-Threaded Publish-Subscribe Architecture for Continuous Queries over Streaming Data Sources. Technical report, Philipps-University Marburg, Department of Mathematics and Computer Science, 2003.

[CJSS03a] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003, June 9-12, 2003, San Diego (CA), USA)*, pages 647–651. ACM, 2003.

[CJSS03b] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. The Gigascope Stream Database. *IEEE Data Engineering Bulletin*, 26(1):27–32, 2003.

[CKSV06] Michael Cammert, Jürgen Krämer, Bernhard Seeger, and Sonny Vaupel. An Approach to Adaptive Memory Management in Data Stream Systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE 2006, April 3-7, Atlanta (GA), USA)*, page 137, 2006.

[CMN99] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. On Random Sampling over Joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1999, June 1-3, Philadelphia (PA), USA)*, pages 263–274, 1999.

[Coc77] William G. Cochran. *Sampling Techniques.* Wilay, New York, 3. edition, 1977.

[Dat00] Christopher J. Date. *An Introduction to Database Systems.* Addison Wesley, 7. edition, 2000.

[DGGR02] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD 1999, June 1-3, Philadelphia (PA), USA)*, pages 61–72, 2002.

[DGR03] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate Join Processing Over Data Streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003, June 9-12, San Diego (CA), USA)*, pages 40–51, 2003.

[FHKS05] Conny Franke, Michael Hartung, Marcel Karnstedt, and Kai-Uwe Sattler. Quality-aware Mining of Data Streams. In *Proceedings of the 10th International Conference on Information Quality (ICIQ 2005, November 4-6, Boston (MA), USA)*, 2005.

[GBLP96] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE 1996, February 26 - March 1, New Orleans (LA), USA)*, pages 152–159, 1996.

[GG02] Minos N. Garofalakis and Johannes Gehrke. Querying and Mining DataStreams: You only Get One Look. (Tutorial). In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002, August 20-23, Hong Kong, China)*, 2002.

[GÖ03a] Lukasz Golab and M. Tamer Özsu. Data Stream Management Issues - A Survey. Technical report, School of Computer Science, University of Waterloo, Canada, 2003.

[GÖ03b] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[GÖ03c] Lukasz Golab and M. Tamer Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003, September 9-12, Berlin, Germany)*, pages 500–511, 2003.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[GV04] Janusz R. Getta and Ehsan Vossough. Optimization of Data Stream Processing. *SIGMOD Record*, 33(3):34–39, 2004.

[Haa97] Peter J. Haas. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *Proceedings of the Ninth International Conference on Scientific and Statistical Database Management (SSDBM'97, August 11-13, Olympia (WA), USA)*, pages 51–63, 1997.

[Haa05] Peter J. Haas. Data-stream sampling: basic techniques and results. *Data Stream Management: Processing High Speed Data Streams. Springer Verlag*, 2005.

[HAE05] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Optimizing In-Order Execution of Continuous Queries over Streamed Sensor Data. In *Proceedings of the 17th International Scientific and Statitical Database Management Conference (SSDBM'05, June 27-29, Santa Barbara (CA), USA)*, pages 143–146, 2005.

[Ham97] Claude-Joachim Hamann. On the Quantitative Specification of Jitter Constrained Periodic Streams. In *Proceedings of the 5th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 1997, January 15-17, Haifa, Israel)*, pages 171–176, 1997.

[HBR+05] Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Ugur Çetintemel, Michael Stonebraker, and Stanley B. Zdonik. High-Availability Algorithms for Distributed Stream Processing. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005, April 5-8, Tokyo, Japan)*, pages 779–790, 2005.

[HFAE03] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003, September 9-12, Berlin, Germany)*, pages 297–308, 2003.

[HHW97] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD 1997, May 13-15, Tucson (AZ), USA)*, pages 171–182, 1997.

[HMA+04] Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, Ann Christine Catlin, Ahmed K. Elmagarmid, Mohamed Y. Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, Robert Gwadera, Ihab F. Ilyas, Mirette S. Marzouk, and Xiaopeng Xiong. Nile: A Query Processing Engine for Data Streams. In *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004, March 30 - April 2, Boston (MA), USA)*, page 851, 2004.

[JC04] Qingchun Jiang and Sharma Chakravarthy. Scheduling Strategies for Processing Continuous Queries over Streams. In *Proceedings of the 21st British National Conference on Databases (BNCOD 21, July 7-9, Edinburgh, UK)*, pages 16–30, 2004.

[JKM+98] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal Histograms with Quality Guarantees. In *Proceedings of 24rd International Conference on Very Large Data Bases (VLDB 1998, August 24-27, New York City (NY), USA)*, pages 275–286, 1998.

[JMSS05] Theodore Johnson, S. Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. A Heartbeat Mechanism and Its Application in Gigascope. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005, August 30 - September 2, Trondheim, Norway)*, pages 1079–1088, 2005.

[KNV03] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating Window Joins over Unbounded Streams. In *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003, March 5-8, Bangalore, India)*, pages 341–352, 2003.

[KS03] Nick Koudas and Divesh Srivastava. Data Stream Query Processing: A Tutorial. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003, September 9-12, Berlin, Germany)*, page 1149. Morgan Kaufmann, 2003.

[KS04] Jürgen Krämer and Bernhard Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 2004, June 13-18, Paris, France)*, pages 925–926, 2004.

[KS05a] Nick Koudas and Divesh Srivastava. Approximate Joins: Concepts and Techniques. (Tutorial). In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005, August 30 - September 2, Trondheim, Norway)*, page 1363, 2005.

[KS05b] Jürgen Krämer and Bernhard Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *Proceedings of the Eleventh International Conference on Management of Data (COMAD 2005, January 6-8, Goa, India)*, pages 70–82, 2005.

[KSKR05] Richard Kuntschke, Bernhard Stegmaier, Alfons Kemper, and Angelika Reiser. StreamGlobe: Processing and Sharing Data Streams in Grid-Based P2P Infrastructures. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005, August 30 - September 2, Trondheim, Norway)*, pages 1259–1262, 2005.

[KSSS04a] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *Proceedings of the 30th International Conference on Very*

Large Data Bases (VLDB 2004, August 30 - September 3, Toronto, Canada), pages 1309–1312, 2004.

[KSSS04b] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB 2004, August 30 - September 3, Toronto, Canada)*, pages 228–239, 2004.

[LS03] Alberto Lerner and Dennis Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003, September 9-12, Berlin, Germany)*, pages 345–356, 2003.

[LZJ+05] Bin Liu, Yali Zhu, Mariana Jbantova, Bradley Momberger, and Elke A. Rundensteiner. A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005, August 30 - September 2, Trondheim, Norway)*, pages 1338–1341, 2005.

[Man03] Paolo Mantegazza. *Dissecting RTAI.* Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano, 2003.

[Mat04] The MathWorks. *Signal Processing Toolbox User's Guide*, 2004.

[MF02] Samuel Madden and Michael J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002, February 26 - March 1, San Jose (CA), USA)*, pages 555–566, 2002.

[MFHH03] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The Design of an Acquisitional Query Processor For Sensor Networks. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003, June 9-12, San Diego (CA), USA)*, pages 491–502, 2003.

[MFHH05] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(1):122 – 173, 2005.

[Mou03] Patrick Mourot. *RTAI Internals Presentation*, 2003.

[MSHR02] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002, June 3-6, Madison (WS), USA)*, pages 49–60, 2002.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Resource Management, and Approximation in a Data Stream Managament System. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003, January 5-8, 2003, Asilomar (CA), USA)*, 2003.

[Nat04] National Instruments Corporation. *E Series Help*, July 2004.

[NS95] Klara Nahrstedt and Ralf Steinmetz. *Multimedia: Computing, Communications & Applications*. Prentice Hall., 1995.

[OFB04] Dan Olteanu, Tim Furche, and François Bry. An efficient single-pass query evaluator for XML data streams. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC 2004, March 14-17, Nicosia, Cyprus)*, pages 627–631, 2004.

[OMFB02] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *In Proceedings of the XML-Based Data Management and Multimedia Engineering Workshop (EDBT 2002 Workshop, March 24-28, Prague, Czech Republic)*, pages 109–127, 2002.

[RDZ+05] Elke A. Rundensteiner, Luping Ding, Yali Zhu, Timothy Sutherland, and Bradford Pielech. CAPE: A Constraint-Aware Adaptive Stream Processing Engine. *Stream Data Management (Advances in Database Systems Series). Springer Verlag*, pages 83–111, 2005.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000, May 16-18, Dallas (TX), USA)*, pages 249–260, 2000.

[SAM98] Sunita Sarawagi, Rakesh Agrawal, and Nimrod Megiddo. Discovery-Driven Exploration of OLAP Data Cubes. In *Proceedings of the 6th International Conference on Extending Database Technology (EDBT'98, March 23-27, Valencia, Spain)*, pages 168–182, 1998.

[Sch01] Rainer Schlittgen. *Zeitreihenanalyse*. Oldenbourg, 9 edition, May 2001.

[ScZ05] Michael Stonebraker, Ugur Çetintemel, and Stan Zdonik. The 8 Requirements of Real-Time Stream Processing. *SIGMOD Record*, 34(4):42–47, December 2005.

[SFL05] Sven Schmidt, Marc Fiedler, and Wolfgang Lehner. Source-aware Join Strategies of Sensor Data Streams. In *Proceedings of the 17th International Scientific and Statitical Database Management Conference (SSDBM'05, June 27-29, Santa Barbara (CA), USA)*, 2005.

[SG84] Julius O. Smith and P. Gossett. A Flexible Sampling-Rate Conversion Method. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'84, March 19 - 21, San Diego (CA), USA)*, pages 112–115, 1984.

[SGG02] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts.* Wiley, 6th edition, 2002.

[SH90] Samuel D. Stearns and Don R. Hush. *Digital Signal Analysis.* Prentice Hall, 2nd edition, 1990.

[SH98] Mark Sullivan and Andrew Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *Proceedings of the 1998 USENIX Annual Technical Conference (USENIX 1998, May 15-19, New Orleans (LA), USA)*, 1998.

[SHB03] David Schleef, Frank Hess, and Herman Bruyninckx. *The Control and Measurement Device Interface handbook.* 2003.

[SKK04] Bernhard Stegmaier, Richard Kuntschke, and Alfons Kemper. StreamGlobe: adaptive query processing and optimization in streaming P2P environments. In *Proceeedings of the 1st international workshop on Data management for sensor networks (DMSN 2004, August 30, Toronto, Canada)*, pages 88–97, 2004.

[SLL05] Sven Schmidt, Thomas Legler, and Wolfgang Lehner. Real-time Scheduling for Data Stream Management Systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05, July 6-8, Palma de Mallorca, Spain)*, 2005.

[SLSL05] Sven Schmidt, Thomas Legler, Sebastian Schaer, and Wolfgang Lehner. Robust Real-time Query Processing with QStream. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005, August 30 - September 2, Trondheim, Norway)*, pages 1299–1302, 2005.

[SMFH01] Mehul A. Shah, Samuel Madden, Michael J. Franklin, and Joseph M. Hellerstein. Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System. *SIGMOD Record*, 30(4):103–114, 2001.

[SMW05] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator Placement for In-Network Stream Query Processing. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2005, June 13-16, Baltimore (MD), USA)*, 2005.

[Sta04] Stanford University. *STREAM: The Stanford Stream Data Manager - User Guide and Design Document.*, 2004.

[Sul96] Mark Sullivan. Tribeca: A Stream Database Manager for Network Traffic Analysis. In *Proceedings of 22th International Conference on Very Large Data Bases (VLDB'96, September 3-6, Mumbai (Bombay), India)*, page 594. Morgan Kaufmann, 1996.

[SW98] Claude E. Shannon and Warren Weaver. *The mathematical theory of communication.* University of Illinois Press, 1998.

[SW04a] Utkarsh Srivastava and Jennifer Widom. Flexible Time Management in Data Stream Systems. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2004, June 14-16, Paris, France)*, pages 263–274, 2004.

[SW04b] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB 2004, August 30 - September 3, Toronto, Canada)*, pages 324–335, 2004.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[TcZ+03] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003, September 9-12, Berlin, Germany)*, pages 309–320, 2003.

[Tea02] RTAI Development Team. *DIAPM RTAI - Beginner's Guide*, 2002.

[Tea06] RTAI Development Team. *RTAI API Documentation*, 2006.

[TMSF03] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. In *IEEE Transactions on Knowledge and Data Engineering*, volume 15, pages 555–568, 2003.

[UF00] Tolga Urhan and Michael J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[VN02] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002, June 3-6, Madison (WS), USA)*, pages 37–48, 2002.

[VNB03] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003, September 9-12, Berlin, Germany)*, pages 285–296, 2003.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 1994, November 14-17, Monterey (CA), USA), pages 1–11, 1994.

[YG02] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.

[ZGTS03] Donghui Zhang, Dimitrios Gunopulos, Vassilis J. Tsotras, and Bernhard Seeger. Temporal and spatio-temporal aggregations over data streams using multiple time granularities. *Information Systems*, 28(1-2):61–84, 2003.

[ZKOS05] Rui Zhang, Nick Koudas, Beng C. Ooi, and Divesh Srivastava. Multiple Aggregations Over Data Streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD 2005, June 13-16, Baltimore (MD), USA)*, pages 299–310, 2005.

[ZRH04] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD 2004, June 13-18, Paris, France)*, pages 431–442, 2004.

[ZSC+03] Stanley B. Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Çetintemel, Magdalena Balazinska, and Hari Balakrishnan. The Aurora and Medusa Projects. *IEEE Data Eng. Bull.IEEE Data Engineering Bulletin*, 26(1):3–10, 2003.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, 17. Oktober 2006

Sven Schmidt