# T U M

I N S T I T U T F Ü R I N F O R M A T I K

# Advanced Data Stream Sharing

Richard Kuntschke Alfons Kemper

TUM-I0836
November 08

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-11-I0836-0/1.-FI


Druck: Institut für Informatik der
Technischen Universität München

# Advanced Data Stream Sharing

Richard Kuntschke Alfons Kemper

Lehrstuhl Informatik III: Datenbanksysteme
Fakultät für Informatik
Technische Universität München
Boltzmannstraße 3, D-85748 Garching bei München, Germany
{richard.kuntschke|alfons.kemper}@in.tum.de

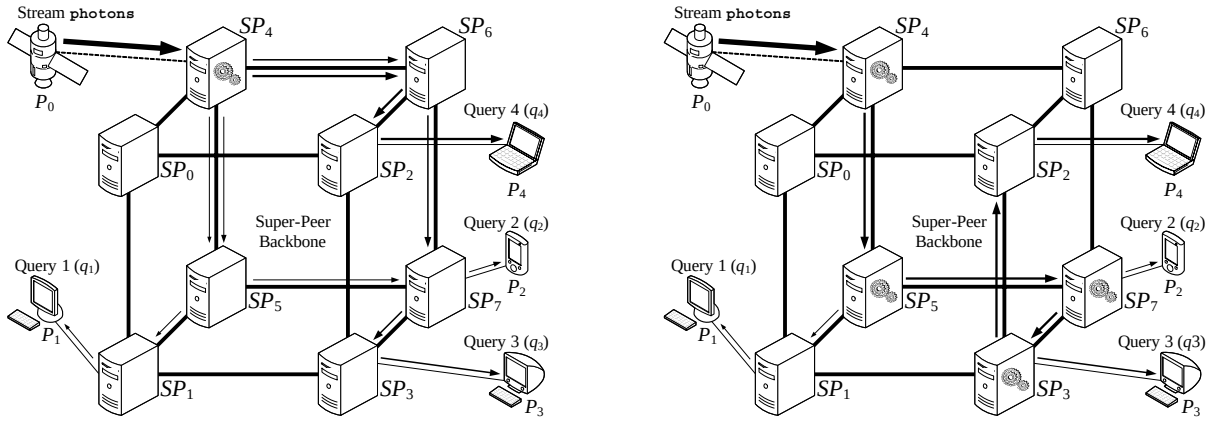
**Abstract**

Using multi-query optimization for sharing common work among multiple queries requires the identification of shareable query components. This kind of optimization is particularly effective in distributed data stream management systems (DSMSs) with multiple continuous queries running concurrently over long periods of time. In this paper, we introduce an abstract property tree (APT) and its extension, an abstract property forest (APF), for representing, matching, and merging XQuery-based queries and XML data streams in a distributed DSMS to enable the sharing of potentially preprocessed data streams among multiple queries. The presented techniques thus allow for efficient resource usage and provide for an increased number of queries that can be processed concurrently.

## 1 Introduction

Deciding whether a query result or a data set contains all the relevant information for answering another query is strongly related to the query containment problem [16] and a common problem in many applications such as view selection [31] and semantic caching [15]. Recently, this problem also arises in data stream sharing in distributed data stream management systems (DSMSs) [29]. In our *StreamGlobe* distributed DSMS [42], a set of super-peers forms a stable grid-based super-peer backbone network with an arbitrary network topology. A super-peer in this context is a stable, powerful server with extensive query processing capabilities that runs a grid middleware and makes its functionality available as a grid service. Thin-peers are usually smaller and possibly mobile devices such as sensors or workstations that can join or leave the network and act as data sources and data sinks. When joining, a thin-peer registers itself at a super-peer in the backbone network. Subsequently, the thin-peer can register data streams and continuous queries—also simply called queries or subscriptions in the following—at its super-peer. The DSMS needs to assure that each query is processed correctly and that the corresponding result data stream is delivered to the peer the respective query is registered at. *Data stream sharing* is an optimization technique for reducing CPU load and network traffic in such a distributed DSMS by means of *in-network query processing*, i. e., distributing query processing operators in the network, and *multi-subscription optimization*, i. e., using one data stream to satisfy multiple similar queries.

We have shown in previous work [29] how data stream sharing can improve resource usage in a distributed DSMS by sharing the preprocessed result data streams of previously registered queries in the network for satisfying newly arriving queries if appropriate. However, the optimization quality of this approach depends on the query registration sequence. Only if a newly registered query requires at most the same data as a previously registered query, sharing the previous result for satisfying the new query is possible. In this paper, we introduce *data stream widening* as an additional technique for making the optimization quality more independent from the query registration sequence and the actual query characteristics. The technique is able to widen an existing data stream to additionally include all the necessary data for the new query. We also devise the inverse *data stream narrowing* for downsizing a data stream in case a dependent query has been deleted from the system. Furthermore, the techniques we introduce in this paper support a larger class of queries. While the previous approach supports flat




**(a)** Data stream sharing without data stream widening

**(b)** Data stream sharing with data stream widening

**Figure 1:** Example DSMS scenario

selection, projection, and aggregate queries, the new approach additionally supports nested queries and joins.

As a motivating example for the application of advanced data stream sharing with data stream widening in StreamGlobe, we introduce an astrophysical e-science application. Consider Figure 1 which illustrates data stream sharing once without and once with data stream widening in an exemplary network. Here, $SP_0$ to $SP_7$ are the super-peers that constitute the super-peer backbone network and $P_0$ to $P_4$ are thin-peers. $P_0$ is a satellite-bound telescope that detects photons and registers a data stream called `photons` at super-peer $SP_4$. This data stream contains real astrophysical data collected during the ROSAT All-Sky Survey (RASS) [45] which we obtained through our cooperation partners from the Max-Planck-Institut für extraterrestrische Physik[1].

In StreamGlobe, we deal with streams of XML data. Stream `photons` complies to a DTD with the structure shown in Figure 2. As its name implies, the data stream delivers a stream of photons detected, e. g., by a satellite's photon detector. Each photon in the data stream is represented by an XML element `photon` that incorporates the coordinates of the corresponding photon (`coord`), the pulse height channel, i. e., the detector pulse caused by the photon when hitting the detector (`phc`), the photon's energy in keV (`en`), and the time of its detection in seconds since the start of the observation (`det_time`). The coordinates consist of the celestial coordinates of the position in the sky where the photon was detected (`cel`) and the coordinates of the detector pixel where the photon actually hit the detector (`det`). Celestial coordinates comprise the right ascension (`ra`) and the declination (`dec`) of a point in the sky, measured in degrees. Detector pixel coordinates simply contain the two-dimensional coordinates of the respective pixel on the detector plain (`dx`, `dy`). Figure 2 shows the DTD of the example data stream `photons` together with its tree structure.

For simplicity, we consider only one single data stream in our example. However, multiple data streams can be registered at one or more super-peers in the network. Also, while each element in the example DTD except for the `photon` element occurs exactly once, more complex DTDs with varying element occurrences ("`?`", "`+`", "`*`", "`|`") are also possible and can be handled accordingly.

Peers $P_1$ to $P_4$ in the example network are devices of astrophysicists used to register subscriptions in the network referencing the available data stream as input. Subscriptions are registered using *WXQuery*, our XQuery-based subscription language that we introduce in detail in Section 2. All queries in our example scenario reference data stream `photons` as their single input. Figure 3 shows Queries 1 ($q_1$) to 4 ($q_4$) of the example scenario.

The `stream` function was newly introduced by us and indicates a possibly infinite data stream used as input to a query. Queries $q_1$, $q_2$, and $q_4$ select an area in the sky that contains the *Vela supernova remnant*. Queries $q_1$ and $q_2$ are window-based aggregate queries returning the average energy of detected

[1] `http://www.mpe.mpg.de`

```
<!ELEMENT photons  (photon*)>
<!ELEMENT photon   (coord, phc, en, det_time)>
<!ELEMENT coord    (cel, det)>
<!ELEMENT cel      (ra, dec)>
<!ELEMENT ra       (#PCDATA)>
<!ELEMENT dec      (#PCDATA)>
<!ELEMENT det      (dx, dy)>
<!ELEMENT dx       (#PCDATA)>
<!ELEMENT dy       (#PCDATA)>
<!ELEMENT phc      (#PCDATA)>
<!ELEMENT en       (#PCDATA)>
<!ELEMENT det_time (#PCDATA)>
```

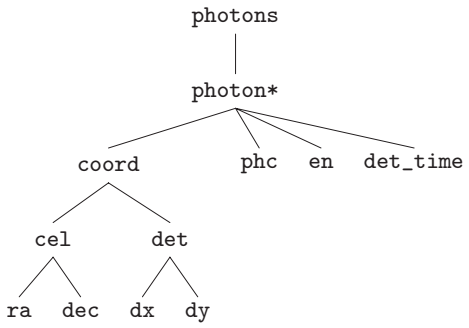



**Figure 2:** DTD of example data stream `photons`

photons in the input stream. While $q_1$ computes the average for all photons with `det_time` values within the last 60 time units and produces an aggregate value every 40 time units, $q_2$ computes the average for all photons with `det_time` values within the last 20 time units and produces an aggregate value every 10 time units. Section 2 presents the details of the window syntax in WXQuery. Furthermore, in contrast to $q_2$, query $q_1$ only returns aggregate values that are greater than or equal to 1.3 keV. Query $q_3$ is a simple selection and projection query delivering the celestial coordinates, the energy, and the detection time of all the photons detected in the area of the *RX J0852.0-4622 supernova remnant* [7] which is situated within the area of *Vela*. Query $q_4$ is similar to $q_3$ but filters the same larger *Vela* section of the sky as $q_1$ and $q_2$. Note that the section of the sky selected by $q_3$ is completely contained in the section selected by $q_4$. Also, $q_3$ is only interested in photons having an energy value of at least 1.3 keV whereas $q_4$ returns information about all the photons in the selected area of the sky and additionally includes the `phc` element in the result.

Assuming that we register queries $q_1$ to $q_4$ one after another in ascending order, data stream sharing without data stream widening is not applicable. The reason is that the later registered queries in this example always need more data than all previous ones. Therefore, multi-subscription optimization has no effect and the optimizer creates and routes a new data stream through the network for each query. Figure 1(a) illustrates this situation.

By using data stream sharing with data stream widening, we are able to alter data streams generated for satisfying previously registered queries to additionally contain all the necessary data for the new query. This yields a larger data stream that constitutes the union of the input data of all dependent queries. We can then replicate the stream at appropriate super-peers in the network and further process each of its copies to form the query result for each dependent query. Figure 1(b) shows the result for our example scenario. Note that now, with the exception of $q_1$ which is registered first, each newly registered query shares the widened result data stream of a previously registered query. The effect can be seen when comparing the number of arrows indicating the data flow in the backbone network in Figures 1(a) and 1(b). Without data stream widening, there are nine arrows in the backbone network, with data stream widening there are only five.

In detail, we make the following contributions in this paper:

- We introduce the *Abstract Property Tree (APT)*, a structure used for representing, matching, and merging queries and data that naturally supports data stream widening and data stream narrowing (Section 3). We focus on queries over XML data streams formulated in our XQuery-based subscription language *WXQuery* (Section 2). We initially consider selection, projection, and aggregate queries and subsequently introduce an extension called *Abstract Property Forest (APF)* to additionally support join queries (Section 5).
- We show how to translate an arbitrary WXQuery into a corresponding APT and how to translate an APT back into a corresponding WXQuery. We define inference rules for the translation of a WXQuery into an APT and query templates for the inverse translation (Section 3). Further, we extend our results to APFs (Section 5).
- We present an algorithm for matching and merging two APTs, yielding a new APT that repre-

```
<photons>
  { for $w in stream("photons")/photons/photon
      [coord/cel/ra >= 120.0 and
       coord/cel/ra <= 138.0 and
       coord/cel/dec >= -49.0 and
       coord/cel/dec <= -40.0]
      |det_time diff 60 step 40|
    let $a := avg($w/en) where $a >= 1.3
    return <avg_en> { $a } </avg_en> }
</photons>
```

**(a)** Query 1 ($q_1$)

```
<photons>
  { for $w in stream("photons")/photons/photon
      [coord/cel/ra >= 120.0 and
       coord/cel/ra <= 138.0 and
       coord/cel/dec >= -49.0 and
       coord/cel/dec <= -40.0]
      |det_time diff 20 step 10|
    let $a := avg($w/en)
    return <avg_en> { $a } </avg_en> }
</photons>
```

**(b)** Query 2 ($q_2$)

```
<photons>
  { for $p in stream("photons")/photons/photon
    where $p/en >= 1.3
      and $p/coord/cel/ra >= 130.5
      and $p/coord/cel/ra <= 135.5
      and $p/coord/cel/dec >= -48.0
      and $p/coord/cel/dec <= -45.0
    return
      <rxj>
        { $p/coord/cel/ra }
        { $p/coord/cel/dec }
        { $p/en } { $p/det_time }
      </rxj> }
</photons>
```

**(c)** Query 3 ($q_3$)

```
<photons>
  { for $p in stream("photons")/photons/photon
    where $p/coord/cel/ra >= 120.0
      and $p/coord/cel/ra <= 138.0
      and $p/coord/cel/dec >= -49.0
      and $p/coord/cel/dec <= -40.0
    return
      <vela>
        { $p/coord/cel/ra }
        { $p/coord/cel/dec }
        { $p/phc } { $p/en } { $p/det_time }
      </vela> }
</photons>
```

**(d)** Query 4 ($q_4$)

**Figure 3:** Example queries

sents the union of the original APTs (Section 4) and generalize the algorithm for use with APFs (Section 5).

- Some preliminary performance experiments conducted using our StreamGlobe prototype implementation assess the effectiveness of data stream sharing with data stream widening based on APTs and APFs (Section 6).

# 2 The WXQuery Language

In StreamGlobe, we use the *Windowed XQuery (WXQuery)* subscription language to register subscriptions over XML data streams. WXQuery is a fragment of XQuery [46] augmented with support for window-based operators. The basic idea and the semantics behind our window extensions are similar to another recent proposal for XQuery window extensions [9].

In Definition 2.1 below, $\alpha$ is a WXQuery expression and $\chi$ denotes a condition. A tag name is denoted by $t$. Further, $\$x$ and $\$y$ are variables representing XML trees, where $\$y$ can also start with a function call to reference a document node or the stream node of a data stream such as `stream("photons")` in the example subscriptions. A variable representing an aggregate result is denoted by $\$a$. The variable $\$z$ can represent any of the three kinds of variables $\$x$, $\$y$, or $\$a$ as described above. We use $\overline{\pi}$ to denote a relative path that only employs the child axis ("`/`"). It does not include wildcards ("`*`"), conditions ("`[`$p$`]`"), or other axes (e. g., "`//`"). A relative path $\pi$ differs from $\overline{\pi}$ in that it can also contain conditions. An aggregate function is denoted by $\Phi$, i. e., $\Phi \in \{\texttt{min}, \texttt{max}, \texttt{sum}, \texttt{count}, \texttt{avg}\}$. In an actual query, each occurrence of the patterns introduced above must be instantiated to an actual object, e. g., each $\alpha$ needs to be instantiated to an actual WXQuery expression and each $\pi$ needs to be instantiated to an actual relative path. Patterns are treated like non-terminals in grammar productions, i. e., multiple occurrences of the same pattern in an expression can and generally will be instantiated to different actual objects. For example, the two occurrences of $\alpha$ in the conditional expression (Expression 4 in Definition 2.1 below) will in general be instantiated to different expressions, one for the `if-then` part and one for the `else` part.

We use a syntax resembling regular expressions to mark optional or recurring parts of a query. Expressions enclosed in $[\![\,]\!]^?$, $[\![\,]\!]^*$, or $[\![\,]\!]^+$ in the definition are optional, can occur zero or more times, or can occur one or more times, respectively. A vertical bar (|) indicates an alternation. An expression of the form $\alpha_{i_1,\ldots,i_n}$ represents a WXQuery expression from a restricted set of expressions. For example, $\alpha_{1,2}$

stands for any one of the two element constructor expressions numbered 1 and 2 in the definition below and $\alpha_{3,4,5,6,7}$ stands for any one of the remaining expressions numbered 3 to 7.

**Definition 2.1 (WXQuery)** The WXQuery subscription language comprises all subscriptions that consist only of the following expressions:

1. <$t$/>
   (empty direct element constructor)
2. <$t$> $[\![\alpha_{1,2}$ | {$\alpha_{3,4,5,6,7}$}$]\!]^*$ </$t$>
   (direct element constructor)
3. $[\![$`for` $\$x$ `in` $\$y[\![/\pi]\!]^?[\![$`|count` $\Delta$ $[\![$`step` $\mu]\!]^?$`|` | `|`$[\![/]\!]^?\overline{\pi}$ `diff` $\Delta$ $[\![$`step` $\mu]\!]^?$`|`$]\!]^?$ |
   `let` $\$a$ `:=` $\Phi$(`$`$\$y[\![/\pi]\!]^?$`)`$]\!]^+$ $[\![$`where` $\chi]\!]^?$ `return` $\alpha$
   (FLWR expression)
4. `if` $\chi$ `then` $\alpha$ `else` $\alpha$
   (conditional expression)
5. $\$y/\pi$
   (output of subtrees reachable from node $\$y$ through path $\pi$)
6. $\$z$
   (output of subtree rooted at node $\$z$)
7. `(`$[\![\alpha[\![$`,`$\alpha]\!]^*]\!]^?$`)`
   (sequence)

□

The FLWR expression in the WXQuery definition introduces our new syntax for expressing data windows, e. g., for use with window-based aggregate operators. The definition of a data window is enclosed in "|" characters. Count-based windows—indicated by the keyword `count`—contain a fixed number of items given by the numeric value of $\Delta$. Optionally, a step size $\mu$ determining the update interval of the data window can be specified. For example, the window `|count 20 step 10|` defines a data window that always contains 20 data items and, during each update, removes the 10 oldest entries from the window while adding the next 10 new data items arriving on the stream. If omitted, the step size defaults to the value of $\Delta$, meaning the contents of the window are completely replaced by new ones during each update.

The situation is analogous for time-based windows except that $\Delta$ indicates the size of the window in time units and the step size indicates the time interval between two successive data windows. Again, the step size defaults to $\Delta$ if omitted. Time-based windows can only be applied on data streams that are sorted according to the values of a particular *reference element* that is used to control the window. This premise could be somewhat relaxed to a fuzzy order by requiring that a fixed sized buffer is sufficient to derive the total order. An example for a time-based window is `|det_time diff 60 step 40|` in query $q_1$. Note that the path inside the window is not meant to be evaluated yielding a sequence as defined by the conventional XQuery semantics. Rather, the path specifies the reference element controlling the window. The path to the reference element is either absolute, starting at the data stream root element (`photons` in our example), or relative to the context node of the data window (`photon` in the example queries).

In the case of subscriptions employing only selection and projection operators, the schema of a data stream generated during in-network query processing can differ from the schema of the corresponding original data stream only by some missing elements which have been removed by a projection operator. Selection operators do not affect the data stream schema at all. Any other more complex data stream schema transformations such as the construction of new elements in the result returned by a query as well as the reordering and renaming of input stream elements in the query result are postponed to a postprocessing step. The postprocessing takes place at the super-peer that is connected to the peer that registered the original subscription. The result of the postprocessing is delivered to its final destination and is not considered for further reuse in the network. The only exception are subscriptions containing aggregate or join operators. In this case, a result data stream with a generic schema is produced by in-network query processing. The generic schema consists of a generic enclosing element for each data

stream item in the result data stream and one generic subelement for each aggregate or join result value computed in the subscription.

Up to now, we restrict the discussion to queries with at most one data window per input data stream. We require each result item returned by a query to contain at least one element of the query input or an aggregate value based on elements of the query input. Thus, we can guarantee that the result of in-network query processing contains all the necessary information for postprocessing. An example for an invalid query would be a query that returns an empty tag for each photon with an energy value above a certain threshold. Since attributes in XML data can always be converted to corresponding elements, we restrict ourselves to dealing with elements. For evaluating continuous WXQueries over XML data streams, we use an extended version of the FluX query engine [22, 23] that supports our window extensions.

# 3 The Abstract Property Tree (APT)

In this section, we define the abstract property tree (APT), a data structure used for representing, matching, and merging queries and data as needed for data stream sharing and data stream widening. We furthermore show how to translate a WXQuery into a corresponding APT and vice versa.

## 3.1 Definition

An *abstract property tree (APT)* consists of two main parts. The first part is a path tree representing all paths referenced in the corresponding query and the second part is a set of annotations. The path tree reflects the *structural* aspects of the query while the annotations reflect its *content-based* aspects, e. g., selection predicates, join predicates, data window definitions, and aggregates. Note that an APT is an *abstract* representation of a query, i. e., it represents only the relevant parts of the query as needed for data stream sharing or, more generally, query result sharing. With the exception of aggregate and join queries, APTs abstract from any complex restructuring of the query result relative to the query inputs as described in Section 2. This abstraction makes the difficult task of matching and merging queries and data feasible in practice.

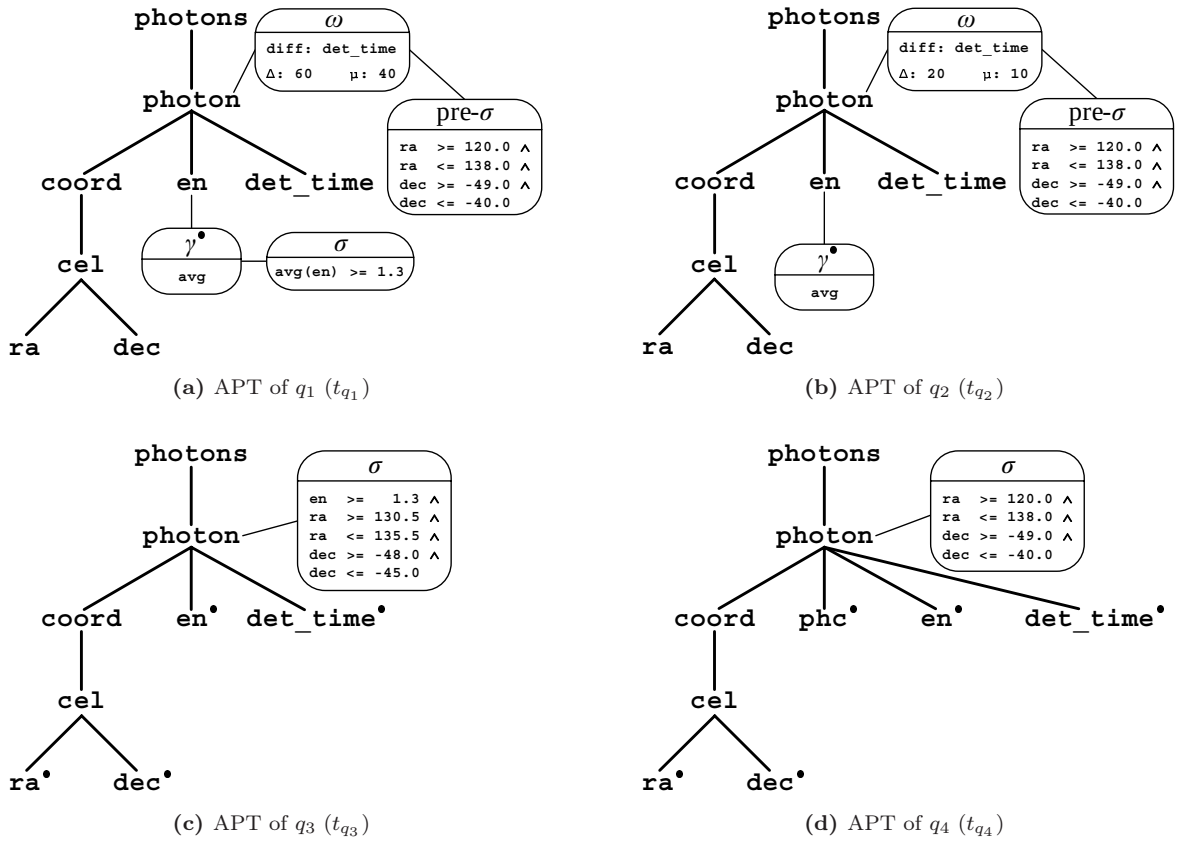
**Definition 3.1 (Query abstraction)** The abstraction $\hat{q}$ of a query $q$ reflects all the properties of $q$ that are relevant for in-network query processing. Compared to the original query $q$, the corresponding abstraction $\hat{q}$ does not contain any query details that are postponed to the postprocessing step, such as any restructuring of the query result involving element construction, reordering, or renaming. Let $q$ be a query, $\mathsf{APT}(x)$ a function that returns the corresponding APT of a query $x$, and $\mathsf{Query}(y)$ a function that returns the corresponding query of an APT $y$. Then, the abstraction $\hat{q}$ of $q$ is obtained as follows:

$$\hat{q} := \mathsf{Query}(\mathsf{APT}(q))$$ □

Figure 4 shows the APTs of the four example queries of Figure 3. The path tree in each case reflects all the paths referenced in the corresponding query. The APT of $q_4$ in Figure 4(d) for example contains the path `/photons/photon/phc` because the `phc` element is returned and therefore referenced in the query. However, the `phc` element does not occur in the APTs of queries $q_1$ to $q_3$ because these queries do not reference this element. Note that all paths referenced in a query are always expanded to absolute paths starting at the data stream root element in the corresponding APT.

The boxes in Figure 4 represent annotations that augment the structural information of the path tree with additional content-based information. There are three types of annotations reflecting the characteristics of the three content-based operators for selection ($\sigma$), window construction ($\omega$), and aggregation ($\gamma$).

Selection annotations are associated either with output elements in the path tree, i. e., with elements that are actually contained in the query result, or with aggregate annotations denoting returned aggregate values. A selection annotation indicates under which condition the corresponding element or aggregate value is returned by the query. Output elements are marked with bullets in an APT. In Figure 4(c), for example, the output elements are `ra`, `dec`, `en`, and `det_time`. Queries returning aggregate values are special since, in their APTs, bullets also mark the aggregate annotations of the aggregate values returned by the query as shown in Figures 4(a) and 4(b). Also, Figures 4(c) and 4(d) indicate that common selection




**(a)** APT of $q_1$ ($t_{q_1}$)

**(b)** APT of $q_2$ ($t_{q_2}$)

**(c)** APT of $q_3$ ($t_{q_3}$)

**(d)** APT of $q_4$ ($t_{q_4}$)

**Figure 4:** APTs of example queries from Section 1

annotations of multiple elements can be pulled up to a common ancestor node. Pulled-up annotations are implicitly considered valid for all output elements further below in the path tree as long as these do not have any other selection annotations associated with them.

Window annotations are always associated with the window root element, i. e., the element whose instances are actually contained in the window. In $q_1$ and $q_2$, `photon` is the window root element. Two different kinds of selection predicates can be associated with window annotations. Predicates in the location steps of a window-defining XPath expression filter the items selected by the XPath expression *before* the items enter the data window. We call selection annotations representing such predicates *window preselection annotations* since the selection takes place before window construction. The symbol pre-$\sigma$ indicates these annotations which appear in the APTs of $q_1$ and $q_2$ in Figure 4. Furthermore, predicates in a `where` condition filter the entire data window *after* it has been constructed in accordance with XQuery existential semantics, treating the window contents as a sequence. We call selection annotations representing these predicates *window postselection annotations* since the selection takes place after window construction. The symbol post-$\sigma$ indicates such annotations.

Finally, aggregate annotations are always associated with the aggregated element, which is `en` in $q_1$ and $q_2$. Like window annotations, aggregate annotations can be associated with two different kinds of selection annotations. An *aggregate preselection annotation* reflects a selection predicate occurring within the XPath expression that references the aggregated element in the argument of the aggregate function call. Such a predicate filters elements *before* the actual aggregate computation, i. e., elements not fulfilling the predicate of the aggregate preselection annotation do not contribute to the aggregate. Furthermore, an ordinary selection annotation associated with an aggregate annotation indicates that the aggregate result value is only returned if the respective condition is satisfied. Figure 4(a) shows such an annotation indicating that the average energy of photons in the specified data window is only to be returned if it is greater than or equal to 1.3 keV. We denote aggregate preselection annotations by the same symbol

pre-$\sigma$ as window preselection annotations. The meaning of the overloaded symbol is unambiguous in an actual APT since the corresponding annotation is either associated with a window annotation or with an aggregate annotation.

For simplicity, we allow window postselection conditions to appear only in the `where` clause of the FLWR expression that defines the corresponding window. Note that element references in annotations are actually absolute paths starting from the data stream root element. In our figures, however, we only show the element name for better readability. Projection operators are structural operators which remove elements from the query inputs. Their effects are therefore already reflected by the path tree. If a query removes elements using a projection, these elements do not appear in the path tree of that query. Thus, there is no projection annotation. We introduce an additional join annotation for representing join operators in Section 5.

**Definition 3.2 (Abstract Property Tree (APT))** The *abstract property tree (APT)* of a query $q$ is denoted $t_q := (P, A, O, id, d)$ and consists of the set of referenced paths $P$, the set of annotations $A$, and the set of returned paths and aggregate values $O$ of $q$, as well as the identifier $id$ and the DTD $d$ of $q$'s input stream or input document.

**Structural part** The set $P$ contains all the paths referenced in the corresponding query. The APT internally represents these paths as a tree with merged common prefixes as shown in Figure 4, i. e., each path element occurs as a tree node exactly once. The tree thus constitutes a prefix tree where each node represents an element occurring in the paths in $P$. A node $v_1$ is the parent of a node $v_2$ in the tree if the element represented by $v_1$ is the parent of the element represented by $v_2$ in a path in $P$. The root of the tree is the root of the query input data stream or document. We expand relative paths referenced in the query to absolute paths before adding them to $P$. The construction of the path tree uses the DTD $d$ to preserve the stream or document order of the elements in the tree. The set $O$ of returned paths and aggregate values identifies the elements in the path tree that we need to mark as output elements. Aggregate values in $O$, indicated by a path with an aggregate function applied to it, cause the corresponding aggregate annotation to be marked with an output marker. As with $P$, we expand all relative paths to absolute paths before adding them to $O$.

**Content-based part** An annotation $a := (\tau, C, R)$ in an abstract property tree has a type $\tau \in \{\sigma, \omega, \gamma, \text{pre-}\sigma, \text{post-}\sigma\}$ indicating a selection annotation, a window annotation, an aggregate annotation, a window preselection annotation or an aggregate preselection annotation, and a window postselection annotation, respectively. The annotation further consists of its contents $C$. In case of a selection annotation, a window preselection annotation, an aggregate preselection annotation, or a window postselection annotation, $C$ is a set of selection predicates. The predicates in the set are meant to be conjunctively combined. A window annotation representing a count-based window contains the window type, the window size, and the step size of the window. In case of a time-based window, the annotation additionally contains the absolute path to the reference element of the window. An aggregate annotation contains the corresponding aggregate function. Finally, $R$ denotes the parents of the annotation, i. e., the objects the annotation is associated with. For selection annotations, $R$ is a set that can contain elements in the path tree as well as aggregate annotations. For window annotations and for aggregate annotations, the parent always is a single element in the path tree. For window preselection and window postselection annotations, the parent always is a window annotation. For aggregate preselection annotations, the parent always is an aggregate annotation. □

## 3.2 Translating WXQueries into APTs

In this section, we show how to translate an arbitrary WXQuery into a corresponding APT.

### 3.2.1 Assembling the Path Tree

We assemble the path tree of a query by first extracting all paths occurring in the respective query. Paths in a query can occur in `for` and `let` clauses, in XPath predicates, in `where` clauses, in window definitions for time-based data windows, in conditional expressions, as parameters of aggregate function

calls, and in return clauses or as standalone path expressions. Each path in a query is either absolute or relative. The query parser extracts all paths occurring in a query and expands each relative path to the corresponding absolute path. In case of paths in XPath predicates and time-based data windows, this is done by concatenating the absolute path of the corresponding context element and the relative path in the predicate or window definition. In all other cases, relative paths start with a variable that can be recursively expanded using a symbol table containing the bindings for all variables in the query.

After extracting all the paths in a query and converting relative to absolute paths, we merge the paths into one path tree. We add the paths to the tree one by one. The process identifies common prefixes which occur in the resulting tree only once. It also preserves the document or stream order. The order of elements on each level of the path tree, from left to right, reflects their order in the query input. Note that up to now, we assume that each query has exactly one input stream. If a query has more than one input stream, we need to build one path tree for each input stream. We extend our solution to this class of queries in Section 5.

**Example 3.1** As an example for path tree assembly consider $q_1$ in Figure 3(a) and its APT $t_{q_1}$ in Figure 4(a). The query contains the absolute path `/photons/photon` and the relative paths `coord/cel/ra` and `coord/cel/dec` in the XPath predicate, `det_time` in the reference element specification of the time-based data window, and `$w/en` as parameter of the aggregate function call. The context element for the XPath predicate and the data window definition is `photon`. Therefore, we expand the relative paths to absolute paths by prepending `/photons/photon`, yielding the absolute paths `/photons/photon/coord/cel/ra`, `/photons/photon/coord/cel/dec`, and `/photons/photon/det_time`. The variable `$w` is bound to a sequence of `photon` elements, i. e. the photons contained in the current data window. We therefore expand the relative path in the aggregate function call by replacing `$w` also with `/photons/photon` yielding `/photons/photon/en` as the final path. When merging the resulting absolute paths into one path tree, we get `/photons/photon` as common prefix of all paths and further `/photons/photon/coord/cel` as common prefix of the two paths in the XPath predicate. The APT of Figure 4(a) contains the resulting path tree. □

### 3.2.2 Determining the Annotations

The next step in APT construction is to determine the annotations. We consider this issue for each of the three main types of annotations, i. e., selection annotations, window annotations, and aggregate annotations, as well as for the three subtypes of selection annotations, i. e., window preselection annotations, aggregate preselection annotations, and window postselection annotations.

**Selection annotations.** We must associate each output element of the path tree and each aggregate annotation representing a returned aggregate value with the condition under which the corresponding element or aggregate value is returned by the query. This condition depends on the context of the respective output element or aggregate value. The relevant conditions can appear as XPath predicates in the location steps of certain XPath expressions, in `where` clauses of FLWR expressions (expression 3 in Definition 2.1), and in conditional expressions (expression 4 in Definition 2.1). Since FLWR expressions and conditional expressions can be nested, the query parser needs to keep track of the current context for each output element. We do this by storing the predicates defined in each FLWR expression or conditional expression in a list and pushing this list on a stack. Whenever an output element is encountered, all predicates in all lists on the stack are conjunctively combined, thus forming the predicate for this element's selection annotation. For conditional expressions, the predicate defined in the expression is used for the `then` part and the negation of this predicate is used for the `else` part. When the scope of a FLWR expression or conditional expression ends, the corresponding predicate list is popped from the stack and will therefore not be part of the selection annotations of subsequent output elements. If the query returns several output elements under the same condition, we try to avoid associating the selection annotation with each output element individually. This is possible by pulling up the selection annotation to a common ancestor node as long as no other output elements with other selection annotations occur between this ancestor node and the output elements.

**Aggregate annotations.** Whenever the query parser discovers a call of an aggregate function, it creates an aggregate annotation indicating the type of the aggregation (`min`, `max`, `sum`, `count`, or `avg`) and associates it with the aggregated element referenced in the aggregate function argument. We associate a corresponding aggregate preselection annotation with the aggregate annotation if the query filters the

sequence of elements to be aggregated prior to aggregation.

**Window annotations.** Whenever the query parser detects a window definition, it creates an according window annotation and associates it with the context element of the window, i. e., the element the window is defined on. Each window annotation contains the window type (count-based or time-based), the reference element (only in case of a time-based window), the window size, and the step size. Optionally, we associate a window preselection annotation, a window postselection annotation, or both with the window annotation if indicated by the query.

We introduce an additional join annotation in Section 5.

**Example 3.2** The APTs of $q_3$ and $q_4$ in Figures 4(c) and 4(d) show examples for selection annotation pull-up. In both queries, all output elements are returned under the same condition. Therefore, the corresponding selection annotation is not associated with each output element individually but pulled up to the first common ancestor node, which is `photon` in both cases.

In the APT of $q_2$ in Figure 4(b), the window annotation is associated with the window context element `photon`. Furthermore, a window pre-selection annotation representing the XPath predicate of the query is associated with the window annotation. Finally, an aggregate annotation marks the `en` element as the aggregated element using an `avg` aggregate. The aggregate annotation also contains an output marker since the corresponding aggregate value is returned by the query. The situation is similar for the APT of $q_1$ in Figure 4(a). The only difference, besides different values in the window annotation, is the additional selection annotation associated with the aggregate annotation. It indicates that the query returns the corresponding aggregate value only under the annotated condition. □

### 3.2.3 Determining the Output Elements

All elements occurring in the path tree of a query are *input elements* of that query, i. e., they must be present in the query input—possibly only under certain conditions expressed by selection annotations. Otherwise, the query will not be answered correctly. The *output elements* of a query are the elements returned by the query, i. e., the elements contained in the query result. Except for aggregate values, each output element also is an input element. However, there can be input elements which are no output elements, e. g., elements that only occur in selection predicates but are never returned by the query. We mark output elements with bullets in APTs as in Figure 4. A special case occurs for queries returning aggregate values. Here, we mark the corresponding aggregate annotations with bullets.

Determining the output elements of a query is a little more difficult than assembling the path tree. The reason is that for building the path tree, we can treat all paths occurring in the query the same. But for determining output elements, we need to decide whether an element referenced in a query $q$ is actually returned by the abstraction $\hat{q}$ of that query. Starting with $q$ as the initial expression $\alpha$, we determine the set of output elements $O_q$ of $q$ recursively as follows. If $\alpha$ is a path expression as in expressions 5 or 6 of Definition 2.1, then add the element referenced by $\alpha$ to $O_q$. If $\alpha$ is a sequence of expressions as in expressions 2 or 7 of Definition 2.1, recursively process each expression in the sequence. If $\alpha$ is a conditional expression as in expression 4 of Definition 2.1, recursively process the expressions in both branches of $\alpha$. If $\alpha$ is a FLWR expression as in expression 3 of Definition 2.1, recursively process the expression returned by $\alpha$.

Currently, we perform the restructuring of the result data stream of structure-preserving queries by applying the original query to the data stream created by in-network query processing. Consequently, we need to assure that each input element required by the original query is present in this stream. We achieve this by additionally marking all input elements of a structure-preserving original query as output elements in the corresponding APT. An optimized approach where elements referenced but not returned by the query are not marked as output elements and remain in the APT only as input elements is possible. This requires rewriting the original query to obtain the correct query for restructuring. The rewriting needs to remove any elements referenced in but not returned by the original query which are no longer needed during restructuring. This can be the case, e. g., because the elements only occur in a selection predicate that has already been evaluated during in-network query processing. The predicate is therefore assured to be satisfied for all remaining data items. This optimization further reduces network traffic for queries for which the set of referenced elements is a proper superset of the set of output elements. Note that this is not an issue for our example queries since $q_1$ and $q_2$ are not structure-preserving and $q_3$

and $q_4$ do not meet the above requirement. Rewriting original queries to generate complex restructuring queries is a matter of future work.

**Example 3.3** In the APTs of $q_1$ and $q_2$ in Figures 4(a) and 4(b), we mark the aggregate annotation with an output marker since these queries return the corresponding aggregate value. The set of output elements of $q_3$ is {`ra`, `dec`, `en`, `det_time`} and that of $q_4$ is {`ra`, `dec`, `phc`, `en`, `det_time`}. Note that, in our current implementation, the set of output elements of $q_3$ would not change if the query would not return the elements `ra`, `dec`, or `en`. Also, the set of output elements of $q_4$ would not change if the query would not return `ra` or `dec`. This is due to the fact that these elements occur in selection predicates of the respective queries. With the optimization described above, however, these elements would be removed from the set of output elements if they were not returned by the query. □

### 3.2.4 Inference Rules

In this section, we introduce formal rules for the translation of a WXQuery into a corresponding APT. There is one rule for each WXQuery expression of Definition 2.1. We use the inference rule notation of the XQuery formal semantics specification [47]. A similar notation has previously been used to describe rules for projecting XML documents to reduce the memory requirements of XML query processors [35, 36]. The judgment

$$Env \vdash \alpha \Rightarrow (P, A, O, id, d)$$

holds if and only if, under the environment $Env$, the expression $\alpha$ references the paths in $P$, defines the annotations in $A$, returns the paths and aggregate values in $O$, and references an input source, i. e., a data stream or a document, with identifier $id$ and DTD $d$. The environment $Env$ holds the symbol table needed for converting relative paths in a WXQuery to absolute paths. Note that all paths are expanded to absolute paths using the variable bindings from $Env$. The set of returned paths $O$ contains absolute paths to returned elements, e. g., `/photons/photon/en`, as well as absolute paths to aggregated elements of returned aggregate values together with the corresponding aggregate function calls, e. g., `avg(/photons/photon/en)`. We determine the input stream identifier or document name $id$ and its corresponding DTD $d$ during a pre-processing phase by scanning the query for any `stream` or `doc` function calls which contain the input source identifier as their parameter. We use the input source identifier to retrieve the corresponding DTD from a metadata repository. Therefore, $id$ and $d$ are already present and simply forwarded in the following rules. Inference rules are of the form

$$\frac{premise_1 \ \ldots \ premise_n}{conclusion}$$

where all premises and the conclusion are judgments of the above form. Additionally, premises may constitute expressions of the form $Env' = Env + (\$var \Rightarrow Path)$ that extend the environment $Env$ by adding the binding of the variable $\$var$ to the path represented by $Path$, thus yielding the extended environment $Env'$. An inference rule expresses that, if all premises hold, then the conclusion holds as well.

We now give the inference rules for each of the WXQuery expressions of Definition 2.1. Since each APT has exactly one identifier $id$ and exactly one DTD $d$, rules 2, 7, and 10 assume that all subexpressions have the same values for $id$ and $d$. As $id$ and $d$ might also be undefined ($\perp$) in certain subexpressions, we implicitly ignore undefined values unless $id$ and $d$ are undefined in all subexpressions of an expression.

**Empty direct element constructor** The empty direct element constructor does not reference or return any paths. It further does not induce any annotations.

$$\frac{}{Env \vdash \texttt{<}t\texttt{/>} \Rightarrow (\emptyset, \emptyset, \emptyset, \perp, \perp)} \quad (1)$$

This inference rule has no premises and therefore, nothing is written above the rule.

**Direct element constructor** The direct element constructor contains zero or more WXQuery expressions. The additions to the APT induced by the direct element constructor are the unions of the additions induced by the enclosed WXQuery expressions. Since an APT always references exactly one input data stream or document, the input identifier $id$ and the DTD $d$ are the same in all expressions, ignoring undefined values as described above.

$$\frac{Env \vdash \alpha_1 \Rightarrow (P_1, A_1, O_1, id, d) \ \ldots \ Env \vdash \alpha_n \Rightarrow (P_n, A_n, O_n, id, d)}{Env \vdash \texttt{<t>}\alpha_1 \ldots \alpha_n\texttt{</t>} \Rightarrow (\bigcup_{i=1}^{n} P_i, \bigcup_{i=1}^{n} A_i, \bigcup_{i=1}^{n} O_i, id, d)} \tag{2}$$

Note that we have rephrased the WXQuery expression for direct element constructors in the inference rule compared to the WXQuery definition to better support the inference rule notation. Although not explicitly shown in the inference rule for simplicity, an expression $\alpha_i$ still needs to be enclosed in curly braces if representing one of the expressions 3 to 7 of Definition 2.1.

**FLWR expression** We split the inference rule for FLWR expressions into four separate rules. Three rules cover `for` loops without data windows and with count-based and time-based data windows, respectively. The fourth rule covers `let` expressions. For better readability, we use shortcuts for certain patterns in the following inference rules. The shortcut $Path_1$ denotes the path $\$y[\![/\pi]\!]^?$ bound to a variable in a `for` loop, $Path_2$ represents the window reference element $[\![/]\!]^?\overline{\pi}$ of a time-based data window, and $Path_3$ stands for the path $\$y[\![/\pi]\!]^?$ in the argument of an aggregate function call.

The $\mathsf{path}$ function used in the inference rules can be applied to any path or aggregate function call. If the argument path is a relative path, the function converts it to the corresponding absolute path. Further, the function removes any conditions from the argument path before returning it. Any aggregate function that is applied to the argument path is preserved by the $\mathsf{path}$ function. The $\overline{\mathsf{path}}$ function can be applied to paths and conditions. It leaves an absolute argument path unchanged and expands a relative argument path to the corresponding absolute path. If the argument path contains any conditions, the paths referenced in these conditions are also extracted, expanded, and returned. The return value of $\overline{\mathsf{path}}$ therefore is a set of paths. When applied to a condition, the $\overline{\mathsf{path}}$ function extracts all the paths referenced in the condition and expands any relative paths to the corresponding absolute paths. When encountering an aggregate function call, the function expands a relative path in the aggregate function argument to an absolute path before returning it. The aggregate function call is removed. The $\mathsf{cond}$ function can be applied to paths and conditions. When applied to a path, it extracts all XPath conditions contained in the argument path. Also, the function expands any relative paths in these conditions to the corresponding absolute paths. The return value of the $\mathsf{cond}$ function therefore is a set of conditions. When applied to a condition, the function expands any relative paths in the condition to absolute paths. Finally, the $\mathsf{id}$ and $\mathsf{dtd}$ functions take a path as argument. If the path starts with a reference to a stream or document node (i. e., with a call to the `stream` or `doc` function), the $\mathsf{id}$ function returns the corresponding stream identifier or document name. The $\mathsf{dtd}$ function uses the stream identifier or document name to retrieve the corresponding DTD of the referenced stream or document from a metadata repository. The stream identifier or document name is read from the argument of the `stream` or `doc` function, respectively. If the argument path does not reference a stream or document node, the $\mathsf{id}$ and $\mathsf{dtd}$ functions return $\perp$. This is safe since we require each query and therefore also each APT to reference exactly one input data stream or document. We deal with queries having multiple inputs in Section 5.

A `for` loop without a window operator references the path bound to the new variable and the paths in the optional XPath and `where` conditions. These conditions also define the selection annotation which is associated with the set of returned paths and aggregate values. If the conditions are not present in the query, the corresponding paths and annotations are not generated. The set of returned paths contains the paths returned by the WXQuery expression $\alpha$ in the `return` clause. The first premise in the rule reflects the variable binding in the `for` loop.

$$\frac{\begin{array}{c} Env' = Env + (\$x \Rightarrow \mathsf{path}(Path_1)) \\ Env' \vdash \alpha \Rightarrow (P, A, O, id, d) \end{array}}{\begin{array}{c} Env \vdash \texttt{for}\ \$x\ \texttt{in}\ Path_1\ \texttt{where}\ \chi\ \texttt{return}\ \alpha \\ \Rightarrow (P \cup \overline{\mathsf{path}}(Path_1) \cup \overline{\mathsf{path}}(\chi), \\ A \cup \{(\sigma, \mathsf{cond}(Path_1) \cup \mathsf{cond}(\chi), O)\}, O, \mathsf{id}(Path_1), \mathsf{dtd}(Path_1)) \end{array}} \tag{3}$$

The above rule reflects the optimized translation of a WXQuery into an APT in the sense described in the previous section on determining the output elements. If the original query should be used for restructuring the resulting intermediate result data stream, then $\overline{\mathsf{path}}(Path_1)$ and $\overline{\mathsf{path}}(\chi)$ need to be added to the set $O$ of returned paths and to the set of parents of the selection annotation that is added to $A$.

The next rule describes the translation of a `for` loop with a count-based data window. The only change compared to the previous rule affects the set of annotations. This set now contains a window annotation for the count-based data window. The window annotation is associated with the element referenced by $Path_1$. Furthermore, we need to break up the selection annotation into a window preselection annotation for the conditions contained in $Path_1$ and a window postselection annotation for the condition in the `where` clause. Both selection annotations are associated with the window annotation $\omega$. The selection annotations are optional, just as the corresponding conditions in the query.

$$\frac{\begin{array}{c} Env' = Env + (\$x \Rightarrow \mathsf{path}(Path_1)) \\ Env' \vdash \alpha \Rightarrow (P, A, O, id, d) \end{array}}{\begin{array}{c} Env \vdash \texttt{for}\ \$x\ \texttt{in}\ Path_1\ \texttt{|count}\ \Delta\ \texttt{step}\ \mu\texttt{|}\ \texttt{where}\ \chi\ \texttt{return}\ \alpha \\ \Rightarrow (P \cup \overline{\mathsf{path}}(Path_1) \cup \overline{\mathsf{path}}(\chi), \\ A \cup \{(\omega, (\text{count}, \Delta, \mu), \mathsf{path}(Path_1)), \\ (\text{pre-}\sigma, \mathsf{cond}(Path_1), \omega), (\text{post-}\sigma, \mathsf{cond}(\chi), \omega)\}, O, \mathsf{id}(Path_1), \mathsf{dtd}(Path_1)) \end{array}} \tag{4}$$

In the same way as in the previous rule, the rule without optimization additionally adds the paths in $\overline{\mathsf{path}}(Path_1)$ and $\overline{\mathsf{path}}(\chi)$ to the set $O$ of returned paths.

The inference rule describing the translation of `for` loops with time-based data windows is similar to the previous rule for count-based windows. The only difference is the additional handling of a path $Path_2$ which identifies the window reference element. The window reference element path occurs in the set of referenced paths and in the window annotation.

$$\frac{\begin{array}{c} Env' = Env + (\$x \Rightarrow \mathsf{path}(Path_1)) \\ Env' \vdash \alpha \Rightarrow (P, A, O, id, d) \end{array}}{\begin{array}{c} Env \vdash \texttt{for}\ \$x\ \texttt{in}\ Path_1\ \texttt{|}Path_2\ \texttt{diff}\ \Delta\ \texttt{step}\ \mu\texttt{|}\ \texttt{where}\ \chi\ \texttt{return}\ \alpha \\ \Rightarrow (P \cup \overline{\mathsf{path}}(Path_1) \cup \overline{\mathsf{path}}(Path_2) \cup \overline{\mathsf{path}}(\chi), \\ A \cup \{(\omega, (\text{diff}, \mathsf{path}(Path_2), \Delta, \mu), \mathsf{path}(Path_1)), (\text{pre-}\sigma, \mathsf{cond}(Path_1), \omega), \\ (\text{post-}\sigma, \mathsf{cond}(\chi), \omega)\}, O, \mathsf{id}(Path_1), \mathsf{dtd}(Path_1)) \end{array}} \tag{5}$$

The rule without optimization additionally adds the paths in $\overline{\mathsf{path}}(Path_1)$, $\overline{\mathsf{path}}(Path_2)$, and $\overline{\mathsf{path}}(\chi)$ to the set $O$ of returned paths.

Finally, the following inference rule defines the translation of `let` expressions which are used to bind the result of an aggregate function call to a variable in WXQuery. The first premise of the rule reflects the binding of the new variable. The rule adds the path $Path_3$ of the aggregated element and, if present, the paths referenced in the condition to the set of referenced paths. It further adds an aggregate annotation to the set of annotations. The aggregate annotation is associated with the aggregated element. Optionally, an aggregate preselection annotation is associated with the aggregate annotation and an ordinary selection annotation is associated with the set of returned elements and aggregate values in $O$.

$$\frac{\begin{array}{c} Env' = Env + (\$a \Rightarrow \Phi\texttt{(}\mathsf{path}(Path_3)\texttt{)}) \\ Env' \vdash \alpha \Rightarrow (P, A, O, id, d) \end{array}}{\begin{array}{c} Env \vdash \texttt{let}\ \$a\ \texttt{:=}\ \Phi\texttt{(}Path_3\texttt{)}\ \texttt{where}\ \chi\ \texttt{return}\ \alpha \\ \Rightarrow (P \cup \overline{\mathsf{path}}(Path_3) \cup \overline{\mathsf{path}}(\chi), \\ A \cup \{(\gamma, \Phi, \mathsf{path}(Path_3)), (\text{pre-}\sigma, \mathsf{cond}(Path_3), \gamma), (\sigma, \mathsf{cond}(\chi), O)\}, \\ O, \mathsf{id}(Path_3), \mathsf{dtd}(Path_3)) \end{array}} \tag{6}$$

In the non-optimized case, the rule additionally adds the paths in $\overline{\mathsf{path}}(Path_3)$ and $\overline{\mathsf{path}}(\chi)$ to the set $O$ of returned paths and consequently also to the set of parents of the selection annotation added to $A$.

**Conditional expression** A conditional expression returns the returned paths and aggregate values of $\alpha_1$ under the condition $\chi$ and those of $\alpha_2$ under the condition $\neg\chi$. The inference rule adds the corresponding selection annotations to the set of annotations $A$. It further adds the paths referenced in the condition to the set of referenced paths $P$. Apart from that, the rule propagates the referenced paths, the annotations, and the returned paths and aggregate values of $\alpha_1$ and $\alpha_2$.

$$\frac{\begin{array}{c} Env \vdash \alpha_1 \Rightarrow (P_{\alpha_1}, A_{\alpha_1}, O_{\alpha_1}, id, d) \\ Env \vdash \alpha_2 \Rightarrow (P_{\alpha_2}, A_{\alpha_2}, O_{\alpha_2}, id, d) \end{array}}{\begin{array}{c} Env \vdash \texttt{if}\ \chi\ \texttt{then}\ \alpha_1\ \texttt{else}\ \alpha_2 \\ \Rightarrow (P_{\alpha_1} \cup P_{\alpha_2} \cup \overline{\mathsf{path}}(\chi), \\ A_{\alpha_1} \cup A_{\alpha_2} \cup \{(\sigma, \mathsf{cond}(\chi), O_{\alpha_1}), (\sigma, \mathsf{cond}(\neg\chi), O_{\alpha_2})\}, O_{\alpha_1} \cup O_{\alpha_2}, id, d) \end{array}} \tag{7}$$

The non-optimized version of the above rule additionally adds the paths in $\overline{\mathsf{path}}(\chi)$ to the set of returned paths $O_{\alpha_1} \cup O_{\alpha_2}$ and to each of the sets of parents of the two selection annotations added to $A_{\alpha_1} \cup A_{\alpha_2}$.

**Output of subtrees reachable from node $\$y$ through path $\pi$** A path expression of this form adds the corresponding path to the sets of referenced and returned paths and generates an additional selection annotation if the path contains predicates. In the inference rule, $Path_4$ represents the pattern $\$y/\pi$.

$$\frac{}{\begin{array}{c} Env \vdash Path_4 \\ \Rightarrow (\overline{\mathsf{path}}(Path_4), \{(\sigma, \mathsf{cond}(Path_4), \{\mathsf{path}(Path_4)\})\}, \{\mathsf{path}(Path_4)\}, \bot, \bot) \end{array}} \tag{8}$$

This rule has no premises.

**Output of a subtree rooted at node $\$z$** The inference rule for this expression adds the path referenced by $\$z$ to the set of returned paths. The path may also contain an aggregate function call. Note that we do not need to add the path to the set of referenced paths since this will be done when processing the expression that defines the variable binding.

$$\frac{}{Env \vdash \$z \Rightarrow (\emptyset, \emptyset, \{\mathsf{path}(\$z)\}, \bot, \bot)} \tag{9}$$

This rule has no premises.

**Sequence** The inference rule for a sequence propagates the union of the sets of referenced paths, annotations, and returned paths and aggregate values of all expressions contained in the sequence.

$$\frac{Env \vdash \alpha_1 \Rightarrow (P_1, A_1, O_1, id, d)\ \ldots\ Env \vdash \alpha_n \Rightarrow (P_n, A_n, O_n, id, d)}{Env \vdash (\alpha_1, \ldots, \alpha_n) \Rightarrow (\bigcup_{i=1}^{n} P_i, \bigcup_{i=1}^{n} A_i, \bigcup_{i=1}^{n} O_i, id, d)} \tag{10}$$

Note that, similar to the rule for direct element constructors, we have rephrased the WXQuery expression for sequences in the inference rule compared to the WXQuery definition to better support the inference rule notation.

**Example 3.4** We use query $q_1$ of Figure 3(a) on page 4 to illustrate the translation of a WXQuery into a corresponding APT following the inference rules introduced above. We start by applying Rules 5 and 6. Note that the four decomposed rules for FLWR expressions always need to be applied in combination since they are actually responsible for handling a single language construct, namely Expression 3 in Definition 2.1 on page 5. We decomposed the rule for FLWR expressions only to make the individual rules more concise.

First, the Rules 5 and 6 update the environment *Env* yielding the extended environment *Env′* by adding $\$w \Rightarrow$ `stream("photons")/photons/photon` and $\$a \Rightarrow$ `avg(stream("photons")/photons/photon/en)`. Using the updated environment whose contents are needed by the path, $\overline{\mathsf{path}}$, and cond functions during the expansion of relative paths to absolute paths, the returned expression `<avg_en> { $a } <avg_en>` is

evaluated next. This is the task of Rule 2 which in turn triggers Rule 9 on the returned variable `$a`. Rule 9 adds the aggregate function call `avg(stream("photons")/photons/photon/en)` to the set of returned paths and aggregate values $O$. The set of referenced paths $P$ and the set of annotations $A$ remain empty. Further, the input stream identifier $id$ and the input stream DTD $d$ remain undefined. Afterwards, Rule 2 simply returns the current state to Rules 5 and 6 for handling the FLWR expression.

Applying Rule 5, $Path_1$ becomes `stream("photons")/photons/photon` and $Path_2$ becomes `det_time`. Further, the value of $\Delta$ is 60 and the value of $\mu$ is 40. The rule adds the following paths to $P$:

- `stream("photons")/photons/photon`
  which corresponds to $Path_1$,
- `stream("photons")/photons/photon/coord/cel/ra`
  and
  `stream("photons")/photons/photon/coord/cel/dec`
  resulting from the condition within $Path_1$,
- `stream("photons")/photons/photon/det_time`
  which is the absolute path of $Path_2$, and
- `stream("photons")/photons/photon/en`
  reflecting the path referenced via $\$a$ in the `where` condition.

The rule further adds to the set of annotations $A$ the window annotation

$$(\omega, (\text{diff}, \texttt{stream("photons")/photons/photon/det\_time}, 60, 40), \texttt{stream("photons")/photons/photon})$$

and subsequently the window preselection annotation

$$\begin{aligned}(\text{pre-}\sigma, \{&\texttt{stream("photons")/photons/photon/coord/cel/ra >= 120.0} \wedge \\ &\texttt{stream("photons")/photons/photon/coord/cel/ra <= 138.0} \wedge \\ &\texttt{stream("photons")/photons/photon/coord/cel/dec >= -49.0} \wedge \\ &\texttt{stream("photons")/photons/photon/coord/cel/dec <= -40.0}\}, \omega).\end{aligned}$$

The set $O$ of output elements remains unchanged to that returned by Rule 2 before. Eventually, $id$ is set to `photons` and $d$ is set to the DTD of stream `photons` shown in Figure 2 on page 3.

Applying Rule 6, $Path_3$ becomes `$w/en`. Subsequently, the rule adds to $P$ the path

$$\texttt{stream("photons")/photons/photon/en}$$

which results from both applications of the $\overline{\mathsf{path}}$ function to $Path_3$ and to the condition $\chi$ in the `where` clause. Further, the rule adds to $A$ the aggregate annotation

$$(\gamma, \texttt{avg}, \texttt{stream("photons")/photons/photon/en})$$

as well as the selection annotation

$$(\sigma, \{\texttt{avg(stream("photons")/photons/photon/en) >= 1.3}\}, \{\texttt{avg(stream("photons")/photons/photon/en)}\})$$

induced by the condition $\chi$ in the `where` clause of the query. Again, $O$ remains unchanged. Since $Path_3$ does not contain a `stream` or `doc` function call, $\mathsf{id}(Path_3)$ and $\mathsf{dtd}(Path_3)$ both return $\perp$.

Figure 4(a) on page 7 shows a graphical representation of the final APT $t_{q_1}$ of $q_1$. □

## 3.3 Translating APTs into WXQueries

The purpose of representing queries using APTs is to abstract from the restructuring details of the query and to enable a feasible way of identifying reusable data streams for data stream sharing. Furthermore, we show in Section 4 how APTs can be merged in order to represent multiple queries, i. e., the union

```
<ROOT>
  { for $VAR in stream("STREAM")/ROOT/ITEM
    return
      if (PRED1 or ... or PREDn) then
        <ITEM>
          ...
          { if (PRED1) then $VAR/PATH1 else () }
          ...
          { if (PREDn) then $VAR/PATHn else () }
          ...
        </ITEM>
      else () }
</ROOT>
```

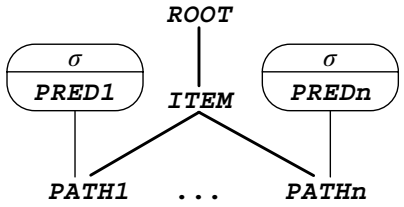



**Figure 5:** Structure-preserving query template and corresponding APT

of the corresponding result data streams, to increase possibilities for data stream sharing. The merged APT then reflects a subscription that can serve as a prefilter for the corresponding original queries. Therefore, each APT represents either the abstraction of a single query or the abstraction of the union of a set of queries. For creating the data streams represented by APTs in a distributed DSMS, we need to install and execute according queries in the system. We distinguish between structure-preserving and structure-mutating APTs.

### 3.3.1 Structure-Preserving APT

A so-called structure-preserving APT represents a query with selection and projection operators but without more complex operators such as window construction and aggregation. We use the query template of Figure 5 for translating such an APT into a corresponding query. We concentrate on queries referencing data streams as input in the following. Queries on documents can be handled analogously. The template contains template variables which are replaced by actual values when generating a query for a given APT. In the template, the template variable *ROOT* stands for the root element of the input data stream (`photons` in our running example), *$VAR* represents an arbitrary variable name, *STREAM* denotes the input data stream (again `photons` in our running example), and *ITEM* references the name of the elements actually contained in the stream (`photon` in the running example). Further, *PRED1* to *PREDn* represent selection predicates, and *PATH1* to *PATHn* represent paths to output elements starting from *$VAR*. These paths can be empty in an actual instance of the template variable, in which case the corresponding preceding slash also disappears from the template.

The replacement of the template variables is straightforward for a given APT except for the predicate template variables *PRED1* to *PREDn*. These represent the predicates of the selection annotations of the APT. The query template returns each output element in the APT under the condition indicated by the corresponding selection annotation. If there is no selection annotation for a certain output element, the query simply returns the element without a surrounding `if` condition. In this case, we also need to remove the `if` condition guarding the output of the *ITEM* tags from the template. The query preserves the stream order, i. e., it returns all elements in the correct order of the data stream schema. We reference an output element in the return clause of the generated query by starting an XPath expression with *$VAR* and concatenating the remaining path steps leading to the output element. The APT yields the paths *PATH1* to *PATHn* by taking the absolute path of the respective output element and removing the prefix bound to *$VAR*. An according prefix replacement also takes place for any paths in the predicates *PRED1* to *PREDn*. The generated query needs to enclose each returned element in the correct sequence of direct element constructors to correctly retain the schema of the original data stream. We can easily derive the necessary information from the paths to the returned elements in the original stream schema. These details vary for each actual query as suggested by the corresponding dots in the template of Figure 5.

**Example 3.5** The APTs of the structure-preserving queries $q_3$ and $q_4$ as shown in Figures 4(c) and 4(d) are translated into the queries of Figures 7(c) and 7(d), respectively. Since the original queries each return all output elements under the same condition as indicated by the selection annotation pull-up in

the APT, only one `if` condition is used in the generated query to return all the output elements. This illustrates how selection annotation pull-up can be used to optimize query generation and reduce query size. In general, if all output elements of a query are returned under the same condition, the `if` condition guarding the output of the *ITEM* element constructor and the `if` conditions guarding the output of the single output elements are all identical. We can therefore leave them all out of the generated query except for the outermost condition which then guards the entire output of the query. □

### 3.3.2 Structure-Mutating APT

A structure-mutating APT represents a window query or an aggregate query. We concentrate on window-based aggregate queries since these are most common in practice and present a query template for aggregate queries with time-based windows. Query templates for aggregate queries with count-based windows, for queries defining data windows without aggregation, and for aggregate queries without data windows look similar. Figure 6 shows the query template for structure-mutating APTs with aggregation and a time-based data window. In addition to the *ROOT*, *$VAR*, *STREAM*, and *ITEM* template variables already known from the template for structure-preserving queries, we introduce the following additional variables. The *PATH* template variable stands for a relative XPath expression with predicates allowed in each location step. We use *REFPATH* to denote a predicateless relative or absolute path. The variable *SIZE* denotes the window size and the variable *STEP* denotes the step size of the data window. The *PRED* variable represents a selection predicate. Further, *AGGVAR1* to *AGGVARn* stand for arbitrary aggregate variable names, *AGGFUNC1* to *AGGFUNCn* each denote one of the aggregate functions `min`, `max`, `sum`, `count`, or `avg`, and *AGGPATH1* to *AGGPATHn* represent paths to the corresponding aggregated element relative to *$VAR*. Moreover, *AGGPRED1* to *AGGPREDn* are optional selection predicates for filtering aggregate values and *AGGELEM1* to *AGGELEMn* are generic aggregate element names. Accordingly, *WINPATH1* to *WINPATHm* denote paths to window elements relative to *$VAR*, and *WINPRED1* to *WINPREDm* are optional selection predicates for filtering window elements. Further, the *WINELEM* template variable represents a generic window root element. The `where` clause, the `if` conditions, and the *PATH*, *AGGPATHi*, and *WINPATHi* variables are optional depending on the characteristics of the corresponding APT. If *PATH* or any *AGGPATHi* or *WINPATHi* is empty in an actual instance of the template variable, the corresponding preceding slash also disappears from the template. If there is no selection annotation for a certain returned aggregate value or window element, the query simply returns the value or element without a surrounding `if` condition. In such a case, we also need to remove any `if` condition guarding the output of the surrounding *ITEM* and *WINELEM* tags from the template.

The query templates for queries defining data windows without aggregation are the same as those for window-based aggregate queries except that the `let` constructs for computing the aggregate values and the corresponding `if` conditions in the return clause are missing. Note that sharing window operators without aggregation during in-network query processing yields no optimization benefit in our setting since we assume potentially overlapping sliding windows that cover the entire input stream. Window operators therefore do not reduce the data volume of the stream. Rather, in case of overlapping windows, the transmitted data volume is increased by repeating the overlapping parts of subsequent windows. The query templates for aggregate queries without data windows are also the same as those for window-based aggregate queries except that the `for` loop, its optional `where` clause, and the window-specific parts in the return clause are missing. The query then needs to reference the input data stream via the `stream` function from within the aggregate function argument. Original queries that contain a `for` loop without a window definition and compute individual aggregate values for each item in the iteration are not meaningful in practice but, for the sake of completeness, are treated internally as if they would define a count-based window with a window size and a step size of one item each. Their APT representation therefore also contains a corresponding window annotation. This is necessary to distinguish such queries from semantically different queries that do not contain any `for` loop and compute a single aggregate value over the entire input. Of course, such queries are only viable on finite inputs.

Again, the determination of the template variable values for a given APT is straightforward. One important issue, however, is that selection predicates in window preselection annotations become XPath predicates in *PATH* whereas selection predicates in window postselection annotations become predicates in *PRED* in a `where` clause. Selection predicates in aggregate preselection annotations become XPath predicates in *AGGPATHi* of the corresponding aggregate function call. We create the generic aggregate

```
<ROOT>
  { for $VAR in stream("STREAM")/PATH|REFPATH diff SIZE step STEP|
    where PRED
    return
    let $AGGVAR1 := AGGFUNC1($VAR/AGGPATH1)
    ...
    let $AGGVARn := AGGFUNCn($VAR/AGGPATHn)
    return
    if (AGGPRED1 or ... or AGGPREDn or WINPRED1 or ... or WINPREDm) then
      <ITEM>
        { if (AGGPRED1) then <AGGELEM1> { $AGGVAR1 } </AGGELEM1> else () }
        ...
        { if (AGGPREDn) then <AGGELEMn> { $AGGVARn } </AGGELEMn> else () }
        { if (WINPRED1 or ... or WINPREDm) then
            <WINELEM>
              { if (WINPRED1) then $VAR/WINPATH1  else () }
                ...
              { if (WINPREDm) then $VAR/WINPATHm  else () }
            </WINELEM>
          else () }
      </ITEM>
    else () }
</ROOT>
```

**Figure 6:** Structure-mutating query template with time-based data window

element name *AGGELEM* by concatenating the actual aggregate function name and the actual name of the aggregated element with an underscore in between, e. g., `avg_en` in our example queries. This is the element name for the aggregate value in the intermediate result data stream generated during in-network query processing. Similarly, we create the generic window root element name *WINELEM* by concatenating a fixed prefix with the name of the actual window root element, e. g., `win_photon`.

We reference both, aggregated elements in the arguments of aggregate function calls as well as output elements in the return clause of the generated query by starting an XPath expression with `$VAR` and concatenating the remaining path steps leading to the respective aggregated or returned element. Note that `$VAR` represents a variable bound to a data window, i. e., to a sequence of elements, in the template of Figure 6. The APT yields the paths *AGGPATH1* to *AGGPATHn* and *WINPATH1* to *WINPATHm* by taking the absolute path of the respective aggregated or returned element and removing the prefix bound to `$VAR`, ignoring the window definition. Again, an according prefix replacement also takes place for any paths in the predicates *AGGPRED1* to *AGGPREDn* and *WINPRED1* to *WINPREDm*.

**Example 3.6** Figures 7(a) and 7(b) show the abstractions of queries $q_1$ and $q_2$ of Figures 3(a) and 3(b), respectively. Note the missing `if` condition guarding the output of the `photon` element constructor in $\hat{q}_2$ compared to $\hat{q}_1$. This is due to the fact that $\hat{q}_2$ does not filter the returned aggregate value and therefore unconditionally produces an output for each data window. We have also optimized $\hat{q}_1$ by removing the `if` condition guarding the output of the `avg_en` element constructor. As in queries $\hat{q}_3$ and $\hat{q}_4$, this is again possible since the query returns elements only under a single condition which is already tested by the surrounding `if` condition guarding the output of the `photon` element constructor. □

# 4 Matching and Merging APTs

We next introduce a tree algebra comprising two operators for matching and merging two APTs. Matching APTs is equivalent to a containment check of the represented query abstractions. We use this for identifying shareable data streams in the network. Merging APTs enables us to compute the union of two queries. This is necessary for data stream widening. Merging also enables data stream narrowing. If several queries depend on the same intermediate data stream generated during in-network processing,

```
<photons>
  { for $w in stream("photons")/photons/photon
      [coord/cel/ra >= 120.0 and
       coord/cel/ra <= 138.0 and
       coord/cel/dec >= -49.0 and
       coord/cel/dec <= -40.0]
      |det_time diff 60 step 40|
    let $a := avg($w/en)
    return
      if ($a >= 1.3) then
        <photon>
          <avg_en> { $a } </avg_en>
        </photon>
      else () }
</photons>
```

**(a)** Abstract Query 1 ($\hat{q}_1$)

```
<photons>
  { for $w in stream("photons")/photons/photon
      [coord/cel/ra >= 120.0 and
       coord/cel/ra <= 138.0 and
       coord/cel/dec >= -49.0 and
       coord/cel/dec <= -40.0]
      |det_time diff 20 step 10|
    let $a := avg($w/en)
    return
      <photon>
        <avg_en> { $a } </avg_en>
      </photon> }
</photons>
```

**(b)** Abstract Query 2 ($\hat{q}_2$)

```
<photons>
  { for $p in stream("photons")/photons/photon
    return
      if ($p/en >= 1.3 and
          $p/coord/cel/ra >= 130.5 and
          $p/coord/cel/ra <= 135.5 and
          $p/coord/cel/dec >= -48.0 and
          $p/coord/cel/dec <= -45.0)
      then <photon>
             <coord>
               <cel>
                 { $p/coord/cel/ra }
                 { $p/coord/cel/dec }
               </cel>
             </coord>
             { $p/en } { $p/det_time }
           </photon>
      else () }
</photons>
```

**(c)** Abstract Query 3 ($\hat{q}_3$)

```
<photons>
  { for $p in stream("photons")/photons/photon
    return
      if ($p/coord/cel/ra >= 120.0 and
          $p/coord/cel/ra <= 138.0 and
          $p/coord/cel/dec >= -49.0 and
          $p/coord/cel/dec <= -40.0)
      then <photon>
             <coord>
               <cel>
                 { $p/coord/cel/ra }
                 { $p/coord/cel/dec }
               </cel>
             </coord>
             { $p/phc } { $p/en }
             { $p/det_time }
           </photon>
      else () }
</photons>
```

**(d)** Abstract Query 4 ($\hat{q}_4$)

**Figure 7:** Abstractions of example queries from Section 1

we potentially can narrow the intermediate data stream by replacing it with the result of merging all remaining queries. If the deleted query required some data that is not needed by any other query, then narrowing will remove this data and the intermediate data stream will therefore become smaller. For large numbers of queries this is, however, expensive. Also, leaving the original intermediate stream in the system might ease sharing for future queries. Therefore, narrowing should only be performed on demand if network bandwidth needs to be freed. It is possible to extend the tree algebra by additional operators. One interesting extension is support for subtraction. Subtracting APTs from each other could for example be used for generating remainder queries in semantic caching [15].

In our application scenario, we always perform matching and merging of APTs in combination. For efficiency reasons, we therefore do the matching and merging in one step by combining both operators in one operation. The operation takes the stream APT and the query APT as input. The stream APT represents the result data stream of a query already installed in the system while the query APT represents a newly arriving query. In the matching step, the matching and merging operation examines whether the data stream represented by the stream APT can be shared for satisfying the query represented by the query APT. If this is not the case, the merging step appropriately merges both APTs, yielding a new APT that represents the union of both input APTs. The resulting APT can be translated into a WXQuery according to Section 3.3. Appropriately installing this query in the system generates a data stream that is shareable by both, the new query and the query represented by the original stream APT. The matching and merging of APTs needs to match and merge the path trees as well as the annotations and the returned elements of the two input APTs.

## 4.1 Matching and Merging the Tree Structures

We match and merge the tree structures of both input APTs by checking whether the path tree of the stream APT contains each path in the path tree of the query APT. If any path is missing, the APTs do not match and we need to merge them. The merging involves adding to the stream APT all the paths of the query APT that are missing in the stream APT. This works just as during path tree construction as described in Section 3.2.1.

There is a special case where we do not need to add all missing elements to the path tree of the stream APT. This case occurs when an ancestor of the subelement to be added is already marked as an output element under the same or a less restrictive condition as the new subelement. In this case, the new subelement is already implicit.

## 4.2 Matching and Merging the Annotations

We match and merge annotations by traversing the APTs and comparing any corresponding annotations, i. e., annotations that are associated with the same elements in both trees, along the way. We need to handle each kind of annotation separately.

**Selection annotations.** For every selection annotation in the stream APT, there must be an according selection annotation in the query APT and the selection predicate in the query APT must imply the predicate of the stream APT. If these conditions are not met, we widen the stream APT by relaxing the selection predicate appropriately, e. g., by forming the union of the stream and the query predicates. We have examined predicate implication checking and relaxation in earlier work [30]. If the query APT contains no selection annotation for a path tree element for which the stream APT does contain a selection annotation, then the widening consists of removing the selection annotation in the merged APT.

**Aggregate annotations.** An aggregate annotation, apart from being associated with the same element, must reference the same aggregate operator in both APTs. Further, we require the predicates of any aggregate preselection annotations to be semantically equivalent. Otherwise, we must remove the aggregate annotation in the merged APT. We must also remove the aggregate annotation if the aggregate is window-based and the corresponding window annotation needs to be removed during merging (see below).

**Window annotations.** The window annotations of the stream APT and the query APT only match if they are defined over the same element in the same data stream, e. g., element `photon` in stream `photons` in our example queries $q_1$ and $q_2$. Further, we require the predicates of any window preselection annotations to be semantically equivalent. The predicates of any window postselection annotation of the window definition in the query APT must imply the predicate of a corresponding window postselection annotation in the stream APT. The window definitions need to fulfill the following conditions for the window size $\Delta$ and the step size $\mu$ of the window definition in the stream APT and the window size $\Delta'$ and the step size $\mu'$ of the window definition in the query APT: $\Delta' \bmod \Delta = 0$, $\Delta \bmod \mu = 0$, and $\mu' \bmod \mu = 0$. Furthermore, the window type (count-based or element-based) must be the same and time-based data windows must have identical reference elements. We have presented more details on sharing window-based aggregate values in previous work [29]. If any of the above requirements is not fulfilled, we remove the window annotation and all dependent aggregate annotations from the merged APT and mark all elements needed by the removed annotations as output elements. We make an exception from this rule for differing window sizes and step sizes of the two data windows. In this case, we perform data stream widening by computing the window size and the step size of a relaxed window. This new window is the basis for a relaxed window annotation which replaces the window annotations of the stream APT and the query APT in the merged APT. The query represented by the resulting APT yields a result data stream that can be used to generate the original data stream as well as to satisfy the new query. The next section details the algorithm for computing the window size and the step size of the relaxed window annotation.

## 4.3 Relaxing Data Windows

The relaxation of data windows works by computing a window size and a step size of a relaxed data window that all dependent windows can share. This requires that, for each dependent window, we can

**Algorithm 1** RelaxWindow

**Input:** Window sizes $\Delta$ and $\Delta'$, step sizes $\mu$ and $\mu'$ of stream and query window, respectively.
**Output:** Window size $\bar{\Delta}$ and step size $\bar{\mu}$ of relaxed window.

1. *Initialize.* Compute the list $L_{\Delta,\Delta'}$ of all common divisors of $\Delta$ and $\Delta'$. Similarly, compute the list $L_{\mu,\mu'}$ of all common divisors of $\mu$ and $\mu'$. These are the sets of potential values for $\bar{\Delta}$ and $\bar{\mu}$, respectively.
2. *Check for compatible pairs.* Iterate over $L_{\Delta,\Delta'}$ and $L_{\mu,\mu'}$ in decreasing order, i. e., examine larger values first. For each $\bar{\mu} \in L_{\mu,\mu'}$ compare $\bar{\mu}$ to each $\bar{\Delta} \in L_{\Delta,\Delta'}$ until the condition $\bar{\Delta} \bmod \bar{\mu} = 0$ is satisfied.
3. *Return result.* Return $\bar{\Delta}$ and $\bar{\mu}$.

combine multiple instances of the relaxed data window to form an instance of the dependent window. Therefore, we do not need to compute the dependent windows or any aggregates on these windows from scratch. Rather, we can determine them by appropriately combining the results of the relaxed window.

The window size and the step size $\bar{\Delta}$ and $\bar{\mu}$ of the relaxed window, and the window and step sizes $\Delta$, $\mu$, $\Delta'$, and $\mu'$ of the stream and the query window, respectively, must satisfy the following conditions: $\bar{\Delta} \bmod \bar{\mu} = 0$, $\Delta \bmod \bar{\Delta} = 0$, $\mu \bmod \bar{\mu} = 0$, $\Delta' \bmod \bar{\Delta} = 0$, and $\mu' \bmod \bar{\mu} = 0$. The task of the window relaxation algorithm therefore is to find suitable values for $\bar{\Delta}$ and $\bar{\mu}$ under the above conditions. Furthermore, to support the optimization goal of reducing network traffic, the resulting data stream should consume as few bandwidth as possible. The major parameter in this respect is the step size. Note that, for example, a window-based aggregate with a count-based data window, a window size of 10, and a step size of 1 causes twice as much network traffic as a window with window size 5 and step size 2. The reason is that the first window produces an aggregate value after *every* data stream item, while the second window produces an aggregate value only after *every second* data stream item.

Algorithm 1 shows how to compute $\bar{\Delta}$ and $\bar{\mu}$ from $\Delta$, $\mu$, $\Delta'$, and $\mu'$. The algorithm takes all potential combinations of $\bar{\Delta}$ and $\bar{\mu}$ into account and chooses the one with the largest value for $\bar{\mu}$ and the largest value of $\bar{\Delta}$ for the chosen value of $\bar{\mu}$ such that the first of the above conditions, which is $\bar{\Delta} \bmod \bar{\mu} = 0$, is satisfied. We choose the largest possible value for $\bar{\mu}$ to minimize network traffic as described above and the largest possible value for $\bar{\Delta}$ for the chosen value of $\bar{\mu}$ to minimize computational effort. Algorithm 1 always finds optimal values for $\bar{\Delta}$ and $\bar{\mu}$. Note that it always finds valid values since, in the worst case, $\bar{\Delta}$ and $\bar{\mu}$ will be set to 1 each.

**Example 4.1** Let $\Delta = 45$, $\mu = 30$, $\Delta' = 30$, and $\mu' = 20$. Then, all three conditions for window shareability as introduced in Section 4.2 are violated:

- $\Delta' \bmod \Delta = 30 \bmod 45 = 30 \neq 0$
- $\Delta \bmod \mu = 45 \bmod 30 = 15 \neq 0$
- $\mu' \bmod \mu = 20 \bmod 30 = 20 \neq 0$

Consider the lists $L_\Delta = [45, 15, 9, 5, 3, 1]$, $L_{\Delta'} = L_\mu = [30, 15, 10, 6, 5, 3, 2, 1]$, and $L_{\mu'} = [20, 10, 5, 4, 2, 1]$ of all divisors of $\Delta$, $\Delta'$, $\mu$, and $\mu'$, respectively. From these lists, Algorithm 1 in the first step determines the list of common divisors of $\Delta$ and $\Delta'$ as $L_{\Delta,\Delta'} = [15, 5, 3, 1]$ and that of $\mu$ and $\mu'$ as $L_{\mu,\mu'} = [10, 5, 2, 1]$. In the second step, the algorithm tests the largest possible value for $\bar{\mu}$, which is 10, against all possible values for $\bar{\Delta}$. This yields the invalid combinations $15 \bmod 10 = 5 \neq 0$, $5 \bmod 10 = 5 \neq 0$, $3 \bmod 10 = 3 \neq 0$, and $1 \bmod 10 = 1 \neq 0$. In practice, the algorithm immediately continues with the next value for $\bar{\mu}$ as soon as the current value of $\bar{\Delta}$ becomes smaller than the current value of $\bar{\mu}$. The algorithm then takes into account the second largest possible value for $\bar{\mu}$, which is 5, and starts again by comparing this value to the largest possible value for $\bar{\Delta}$, which is 15, immediately arriving at the first valid combination $15 \bmod 5 = 0$. In the third step, the algorithm returns the final result $\bar{\Delta} = 15$ and $\bar{\mu} = 5$.

Figure 8 illustrates the correlations between the window sequences of, from top to bottom, the stream window, the query window, and the relaxed window for the above example. The individual shading of the relaxed windows indicates whether a particular relaxed window is shared for building a stream window

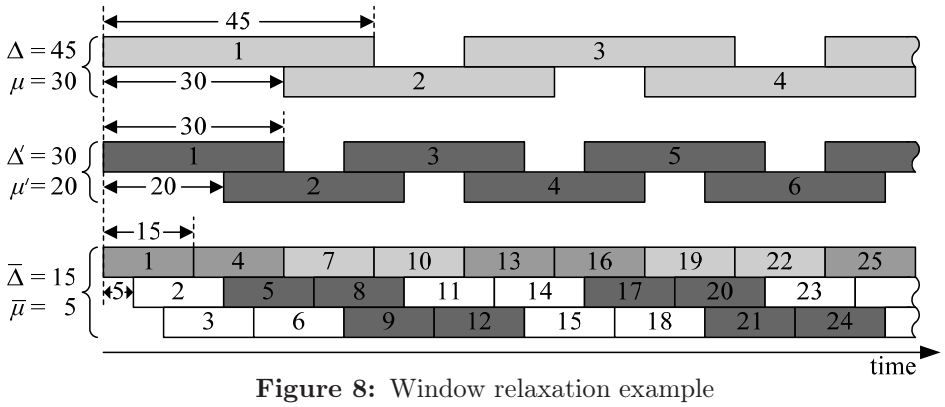



**Figure 8:** Window relaxation example

(light gray), a query window (dark gray), or both (medium gray). Unshaded windows are not shared for any of the two. □

### 4.4 Example Matchings

Consider the APTs of the four example queries in Figure 4. Assuming that the APT of $q_1$ is the query APT and the APT of $q_2$ is the stream APT, applying the rules described above yields a match without widening. If we interchange the roles of the query APT and the stream APT, i. e., match the APT of $q_2$ with the APT of $q_1$, the APTs do not match and need to be merged. The resulting APT is identical with that of $q_2$.

The situation is analogous for the APTs of $q_3$ and $q_4$. Again, matching the APT of $q_3$ with the APT of $q_4$ yields a match without widening since the path tree of $q_3$ contains all paths in the path tree of $q_4$ and the selection predicate of $q_3$ implies the selection predicate of $q_4$. When interchanging the roles of $q_3$ and $q_4$, we have no match since the path tree of $q_3$ does not contain the `phc` element and the inverse implication between the selection predicates is not true. Therefore, we need to merge the APTs, adding the `phc` element and relaxing the selection predicate in the process. The resulting APT is semantically equivalent to the APT of $q_4$, i. e., both APTs represent the same data stream.

Matching the APT of $q_3$ with the APT of $q_1$ leads to the removal of the window annotation and the aggregate annotation together with its associated selection annotation in the merged APT. The window pre-selection annotation becomes a selection annotation associated with the `photon` element and all elements at the leaves of the path tree are marked as output elements. The resulting APT therefore looks similar to the APT of $q_3$. The only difference is in the selection predicate of the selection annotation. Interchanging the roles of the queries here and matching the APT of $q_1$ with the APT of $q_3$ leads to the same result. In this case, the selection predicate of $q_3$ needs to be relaxed and becomes semantically equivalent to the window pre-selection predicate in $q_1$.

## 5 Handling Join Queries

In the following, we extend our findings on APTs from the previous sections to additionally support join queries. Join queries are queries that either reference multiple inputs or that reference the same input multiple times in case of a self-join. Therefore, for each individual input, the abstract property representation of the query contains an individual APT describing the referenced and returned parts of the corresponding input source. Consequently, we call the resulting abstract property representation of such a query an *abstract property forest (APF)*. If inputs are combined, i. e., joined, their respective APTs are interconnected using a new kind of annotation, called a *join annotation*. We begin by introducing our notion of join and query semantics. Then, we describe how APFs are defined on the basis of APTs. Finally, we extend the previously introduced algorithm for matching and merging APTs to support the matching and merging of APFs. Hence, the extensions presented in this section enable the sharing, widening, and narrowing of join query results.

## 5.1 Preliminaries

Before describing the extensions for handling join queries, we first introduce our notion of join and query semantics.

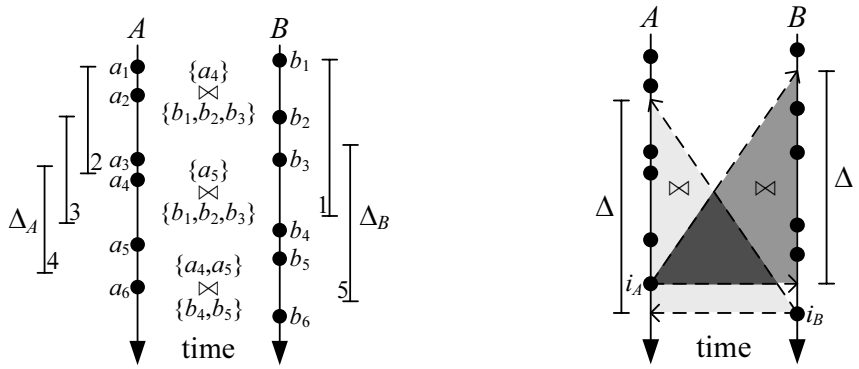
### 5.1.1 Join Semantics

Considering a window-based binary join on two input streams, we define the join semantics as follows. Whenever one of the windows is updated, i. e., the window slides along by the extent defined by its step size, all items entering the window during the update are joined with the contents of the current data window of the other input stream. Consequently, newly arriving data items need to be buffered until the next update is triggered. In case of a count-based data window, the update is triggered after as many items as indicated by the window's step size have arrived on the stream. In case of a time-based data window, the update is triggered when the first item is encountered in the input stream whose reference element value is larger than the projected new upper bound of the window. Due to the sort order of the stream, we can be sure that no more items fitting into the updated window will arrive afterwards.

Whenever a window update occurs, the new items entering the updated window are joined with the current contents of the window of the other input stream. Afterwards, the updated window slides along, removing invalidated items from the window and adding the newly arrived ones. This process easily generalizes to multi-way joins by appropriately joining the new items of the updated window with the current contents of the windows of all other join inputs [18]. For simplicity, we only consider binary joins here.

The step-based join semantics introduced above leads to non-deterministic join results. This is due to the fact that the join result depends on the arrival sequence of data items on the joined input streams. Figure 9(a) illustrates this issue. We assume that the data windows are generated in the sequence indicated by the numbers next to the window intervals in the figure, i. e., the initial window of stream $B$ arrives before the three windows of stream $A$. Finally, the second window of stream $B$ arrives. Note that the time axes in the figure indicate the timestamp values contained in the arriving data items. These represent application time and are independent of the actual arrival time of the data items in the data window. We further assume that the contents of the initial windows of streams $A$ and $B$ in Figure 9(a) have already been joined appropriately. We now consider joins triggered by subsequent window updates. This leads to the three joins indicated in the figure. First, when updating the window over stream $A$, the new parts of the windows numbered 3 and 4, respectively, are joined with the contents of the window numbered 1. This corresponds to the first two joins of the data items $a_4$ and $a_5$ with the data items $b_1$, $b_2$, and $b_3$ in the figure. Subsequently, the new part of the window numbered 5 consisting of $b_4$ and $b_5$ is joined with the complete contents of the window numbered 4 comprising $a_4$ and $a_5$. We can see that a change in the arrival sequence of the windows of both streams—which depends on the arrival sequence of the data items on both streams—can lead to a different join result. For example, if the two windows of stream $B$ arrive between the first and the second window of stream $A$, then $a_4$ and $a_5$ entering the data window of stream $A$ during its first and second update would never be joined with $b_1$ and $b_2$ contained in the first window of stream $B$ in our example. This is different from the window sequence shown in Figure 9(a).

Despite its non-determinism, we make the case for this join semantics. In a multitude of application domains, joining most recent data instead of computing purely timestamp-based joins is of great importance. Prominent examples comprise sensor monitoring, surveillance, traffic supervision, logistics, and process automation control. All of these application scenarios have in common that they need to quickly recognize and react to the newest developments and to exceptional events such as unusual sensor readings, alarms, traffic jams, or malfunctions. Thus, in many cases it is not of primary importance to join data items that have been generated at about the same time and to produce deterministic join results. Instead, it is more important to join the latest, most current values that have arrived on the input streams in order to get the most up-to-date combinations. Our step-based join semantics supports this requirement as long as windows have reasonably small step sizes, e. g., one data item for count-based windows in the extreme case. In the business world, SAP Executive Board member Claus Heinrich has coined the term Real World Awareness [20], emphasizing the importance of monitoring and reacting to most recent data for corporate success. One of the main enabling technologies in this direction is Radio Frequency




**(a)** Step-based **(b)** Traditional

**Figure 9:** Window join semantics

IDentification (RFID). In logistics, for example, reading RFID tags generates streams of events that need to be processed. As a more concrete example, consider stock exchange tickers. When joining the tickers of two companies to compare their relative performance, it is imperative to always combine the latest available values. Since only the most current results are of interest, the fact that the overall join result depends on the arrival sequence of data items is irrelevant. A similar example is to compare the relative performance of the same company at different stock exchanges. In this case, each stock exchange provides one of the input data streams to be joined and the join predicate checks for equality of the company id, assuming that each ticker provides data about multiple companies. The example join queries of Figure 10 stick to our astrophysics application scenario. In this scenario, combining measurements from multiple photon detectors of various telescopes and satellites provides for another possible application of our join semantics. For example, it might be interesting to join photons detected in the same celestial area, i. e., having similar celestial coordinates, and to retrieve their energy and detection time for comparison. For brevity and clarity of exposition, the actual example queries of Figure 10 use simplified join conditions. However, our approach also works for more complex join queries. Another advantage of our join semantics is that no synchronization between join input streams is necessary since we correlate the streams based on their local window definitions which solely depend on the respective input stream. We assume that newly arriving data items from both input streams are processed sequentially to guarantee the synchronization of window updates and associated join computations. Furthermore, the problem of large and growing operator states that requires the introduction of heartbeats or punctuations [44] to limit memory usage when joining slow or bursty input streams is not an issue in our join semantics.

Note that WXQuery can also support different variants of traditional window join semantics over data streams as found in the literature[2]. One of these variants, for example, specifies that each newly arriving data item from one stream is joined with all the data items arriving on the other stream whose timestamps are contained in a certain interval around the timestamp of the new data item. Figure 11 shows an according example WXQuery with $\Delta = 10$. Streams `photon1` and `photon2` are supposed to be photon data streams of the same schema as introduced in Figure 2 on page 3 in all our example join queries. The above semantics has the advantage of producing deterministic join results when using time-based data windows. Count-based data windows always lead to non-deterministic join results in all the join semantics introduced in this section. Efficient join result sharing for join queries using another variant of time-based window join semantics has already been studied extensively [19]. In this variant, data items receive their timestamp on arrival at the join operator. Each data item arriving on an input stream is joined with all data items of the other input stream that arrived previously within a certain time interval. Consider Figure 9(b) that shows an illustrative example. The newly arriving data item $i_A$ with timestamp value $t_{i_A}$ in stream $A$ is joined with all data items of stream $B$ which have arrived previously and whose timestamp values are greater than or equal to $t_{i_A} - \Delta$, with $\Delta$ being the common window size of streams $A$ and $B$. Since each newly arriving data item $i_B$ with timestamp value $t_{i_B}$ in

[2]See, for example, [12, 18, 19, 21, 25, 34].

```
<photons>
  { for $x in stream("photon1")/photons/photon
      |det_time diff 10 step 5|
    for $y in stream("photon2")/photons/photon
      |det_time diff 20 step 10|
    where $x/en >= $y/en + 0.5
    return
      <result>
        { $x/en } { $x/phc }
        { $y/en } { $y/phc }
      </result> }
</photons>
```

**(a)** Query 5 ($q_5$)

```
<photons>
  { for $x in stream("photon1")/photons/photon
      |det_time diff 10 step 5|
    for $y in stream("photon2")/photons/photon
      |det_time diff 20 step 10|
    where $x/en >= $y/en
    return
      <result>
        { $x/en } { $x/det_time }
        { $y/en } { $y/det_time }
      </result> }
</photons>
```

**(b)** Query 6 ($q_6$)

```
<photons>
  { for $x in stream("photon1")/photons/photon
      |det_time diff 30 step 5|
    for $y in stream("photon2")/photons/photon
      |det_time diff 15 step 10|
    where $x/en >= $y/en
    return
      <result>
        { $x/en } { $x/det_time }
        { $y/en } { $y/det_time }
      </result> }
</photons>
```

**(c)** Query 7 ($q_7$)

```
<photons>
  { for $x in stream("photon1")/photons/photon
      |det_time diff 15 step 10|
    for $y in stream("photon2")/photons/photon
      |det_time diff 30 step 15|
    where $x/phc >= $y/phc
    return
      <result>
        { $x/en } { $x/det_time }
        { $y/en } { $y/det_time }
      </result> }
</photons>
```

**(d)** Query 8 ($q_8$)

**Figure 10:** Example join queries

```
<photons>
  { for $x in stream("photon1")/photons/photon
    for $y in stream("photon2")/photons/photon
    where $x/det_time - $y/det_time <= 10
       or $y/det_time - $x/det_time <= 10
    return
      <result>
        { $x/en } { $x/phc }
        { $y/en } { $y/phc }
      </result> }
</photons>
```

**Figure 11:** WXQuery with traditional join semantics

stream $B$ is accordingly joined with all data items of stream $A$ which have arrived previously and whose timestamp values are greater than or equal to $t_{i_B} - \Delta$, $i_A$ will eventually be joined with all data items $i_B$ from stream $B$ for which $(t_{i_A} - t_{i_B} \leq \Delta) \vee (t_{i_B} - t_{i_A} \leq \Delta)$ holds.

The results of [19] are applicable without any changes in our setting when the corresponding join semantics is applied. Note that the optimizations introduced by [19] impose restrictions on the queries taken into account for join result sharing. These restrictions include identical signatures of the join queries, i. e., identical join predicates, and an equal window size $\Delta$ for all input streams of a query as indicated in Figure 9(b). In contrast, our step-based semantics and the accompanying join sharing approach introduced further below allow for different join predicates in the queries taken into account for sharing. We also support queries with varying window and step sizes in the windows of their various input streams.

### 5.1.2 Query Semantics

In SQL, joins can simply be formulated by referencing the relations to be joined in the `from` clause and by including the join predicates as conditions in the `where` clause. The query does not imply a certain

evaluation strategy for computing the join. Therefore, SQL-based continuous query languages such as CQL [5] extend the query language by introducing window syntax constructs without having to change the basic underlying SQL query semantics.

In XQuery and consequently also in WXQuery, joins are expressed by nested `for` loops with accompanying conditions reflecting the join predicates. The usual semantics of nested loops is, however, not applicable when formulating window-based joins over possibly infinite data streams since this leads to infinite loops that do not produce the desired results. To illustrate this issue, consider Query 5 ($q_5$) of Figure 10(a). Both `for` loops in the query reference unbounded data streams with data windows defined on them. Under conventional XQuery semantics, the inner loop would iterate indefinitely over an infinite number of windows on stream `photon2` while the outer loop would never leave its first iteration. Therefore, we redefine the query semantics for join queries in WXQuery as follows. Whenever a WXQuery contains more than one `for` loop over a windowed input, we compute the corresponding window join as described in Section 5.1.1. During join computation in combination with a window update, we consider the variables bound by the `for` loops to iterate over the new items of the updated data window and the current items of the other data window in a nested loops fashion. Due to this change in semantics, we currently do not deal with queries mixing aggregates and joins. Introducing a dedicated WXQuery syntax extension for expressing window-based joins over unbounded data streams is an issue of future work.

## 5.2 The Abstract Property Forest (APF)

In the following, we define APFs and show how to translate a join query into a corresponding APF and vice versa.

### 5.2.1 Definition

The definition of an APF builds on Definition 3.2 of an APT.

**Definition 5.1 (Abstract Property Forest (APF))** We define an *abstract property forest (APF)* $f_q := T$ of a query $q$ with $m$ input data streams as a list $T := [t_q^i \mid 1 \leq i \leq m]$ of property trees, one for each input source referenced in $q$.

**Structural part** The structural part of $f_q$ consists of the union of the structural parts of the contained APTs, i. e., it is a forest consisting of the path trees of the input sources. If a query references the same input source multiple times, e. g., for self-join purposes, then each reference has its own path tree in $f_q$.

**Content-based part** In addition to the annotations of the APTs as introduced in Section 3, $f_q$ can also contain *join annotations*. A join annotation $a := (\tau, C, R)$ is a selection annotation that is associated with elements from multiple APTs. It consists of its type $\tau = \bowtie$, its contents $C$ which represent a set of join predicates, and a set $R$ of parents. Similar to selection annotations, the predicates in the contents $C$ of a join annotation are meant to be conjunctively combined. A join annotation can be associated with elements from each participating APT. As a special kind of a selection annotation, a join annotation is associated with the returned elements of a query and determines under which condition these elements are returned as part of the join result. □

### 5.2.2 Translating WXQueries into APFs

We extend the translation rules introduced in Section 3.2.4 to support join queries referencing multiple input streams.

**Determining Join Annotations** Generating the APTs for each of the multiple input sources, i. e., assembling the path trees, determining the annotations, and identifying the output elements, works exactly as described in Section 3.2 for each input source. The only additional aspect is the identification of join annotations. This is similar to determining selection annotations. If the corresponding predicate is a join predicate, i. e., the predicate correlates elements from different input sources, a join annotation

is generated and associated with the returned elements of the involved sources' APTs. Determining the input source an element belongs to is straightforward since the path to each element is expanded to the corresponding absolute path if necessary. The absolute path contains the respective stream or document identifier as an argument to the `stream` or `doc` function.

**Example 5.1** Figure 12 shows the APFs of the example join queries $q_5$ to $q_8$. Note the join annotations connecting the returned elements of both input streams in each APF. □

The inference rules for translating a WXQuery into a corresponding APF and the query template for translating an APF back into a corresponding WXQuery can be found in the appendix.

## 5.3 Matching and Merging APFs

This section describes how to match and merge APFs for data stream sharing and data stream widening. It further discusses possibilities for join result sharing.

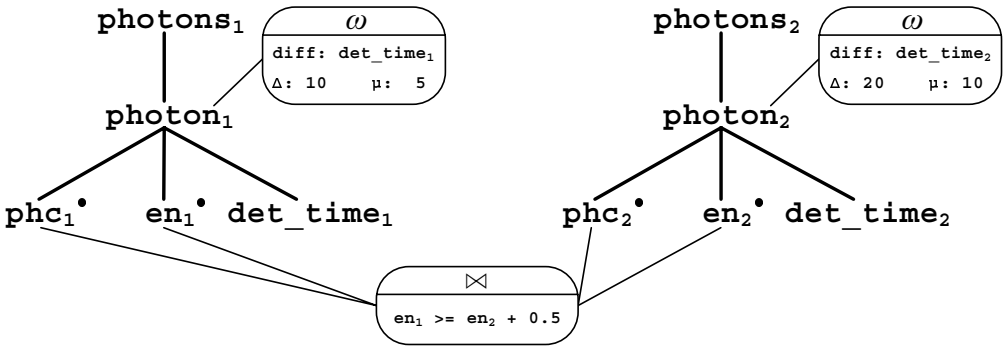
### 5.3.1 Basics

As with APTs, we perform the matching and merging of APFs in one operation which takes two APFs as input, the stream APF and the query APF. The stream APF represents the result data stream of a join query already installed in the system while the query APF represents a newly registered join query. In the matching step, the matching and merging operation examines whether the data stream represented by the stream APF can be shared for satisfying the query represented by the query APF. If this is not the case, the merging step appropriately merges both APFs, yielding either a new APF or—if the join needs to be removed during the merge—a set of APTs representing the necessary inputs for both original APFs. The resulting APF or APTs can accordingly be translated into one or more WXQueries. Appropriately installing these queries in the system generates one or more data streams that are shareable by both, the new query and the query represented by the original stream APF. The matching and merging of APFs needs to match and merge the path trees as well as the annotations of both input APFs.

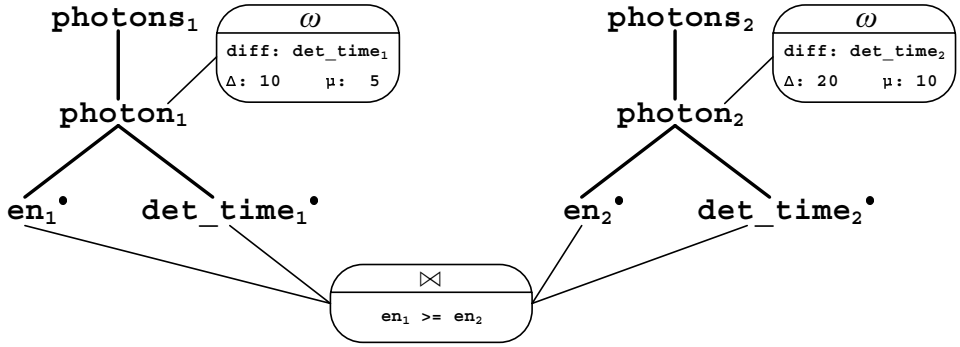
Note that it is also possible to match and merge the APF of a multiple input join query with the APT of a single input non-join query. If the single input query should share the result of the join query, widening will involve removing the join from the merging result. This leads to a set of independent APTs that represent single input queries to be installed in the system. The result data streams of these queries can then be combined later to form the original join result while a copy of one of the streams can further be used to satisfy the new single input query. If the APF of a newly arriving join query is matched with the APT of an already installed single input query, it might be possible to use the result stream of the single input query as one of the inputs to the join query. To determine this, the APT of the corresponding input stream contained in the APF of the join query needs to be matched and merged with the APT of the query whose result shall be shared. This process can be repeated for each input stream of the join query to find suitable streams for all inputs. An open question to be dealt with in future work is where to place the join operator in the network to combine the inputs and to compute the actual join result. Network-aware operator placement has already been the subject of some research work [2, 37, 41]. A simple solution is to route all input streams to the final super-peer which is connected to the peer that registered the new query and to compute the join there. This approach is beneficial if the join result stream is larger than the sum of the sizes of all input streams. Another possibility is to compute the join at any super-peer at which one of the shareable inputs has been found and to route the remaining inputs there. The join result can subsequently be routed to the querying peer. This approach may be beneficial if the join result size is smaller than the sum of the sizes of the join inputs. More sophisticated solutions would make dynamic decisions, e. g., based on statistics and join result size estimations.

### 5.3.2 Sharing Join Results

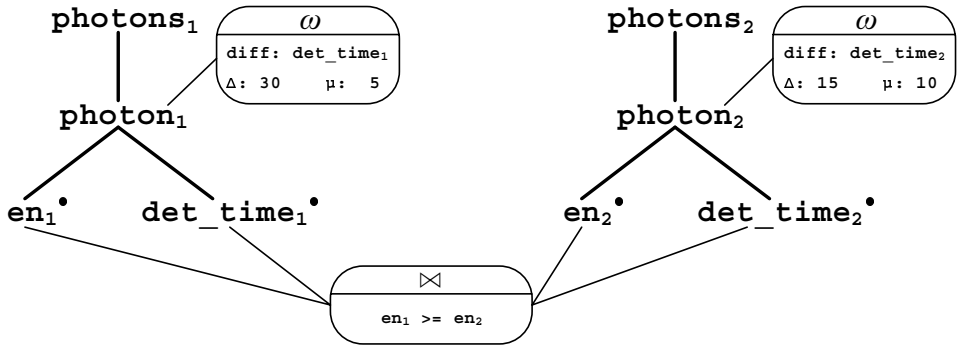
In the following, we concentrate on the matching and merging of the APFs of two binary join queries. Using the join and the query semantics introduced in Section 5.1, we distinguish three cases that allow for different levels of sharing. The cases differ in the relation between the window sizes $\Delta$ and $\Delta'$ as well



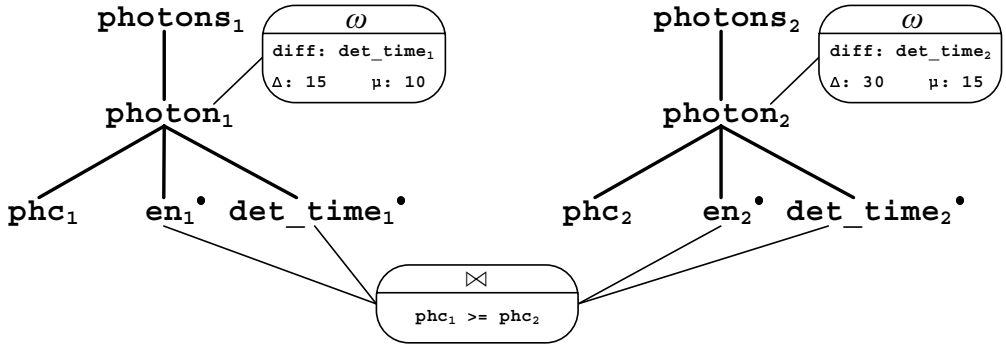

**(a)** APF of $q_5$ ($f_{q_5}$)




**(b)** APF of $q_6$ ($f_{q_6}$)




**(c)** APF of $q_7$ ($f_{q_7}$)




**(d)** APF of $q_8$ ($f_{q_8}$)

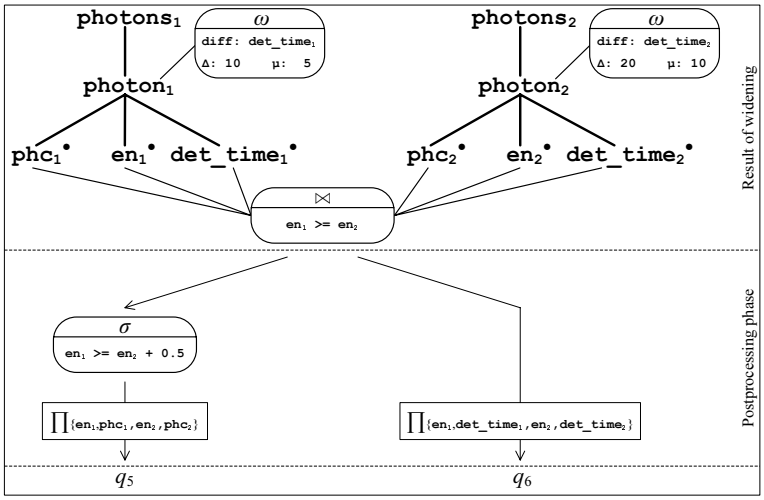
**Figure 12:** APFs of example join queries

as the step sizes $\mu$ and $\mu'$ of each of the data windows in the properties of an already installed query whose result data stream is considered for sharing and a newly arriving query, respectively.

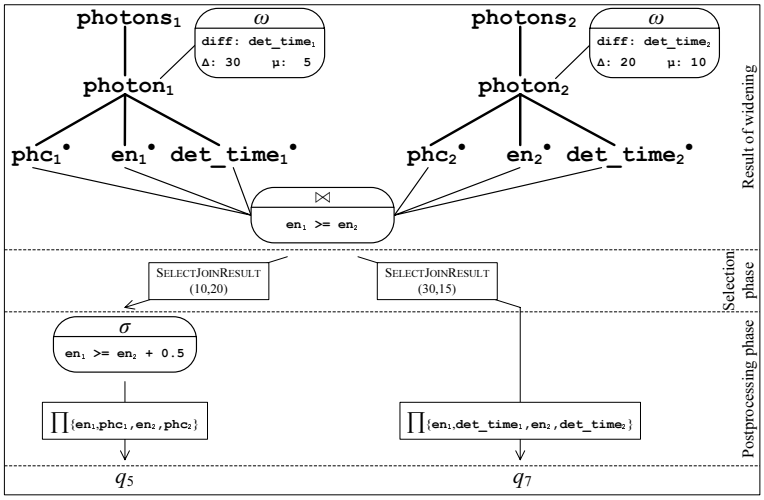
**Full join result sharing** Full join result sharing is the simplest and most effective case. It occurs if $\Delta = \Delta'$ and $\mu = \mu'$ for each pair of corresponding data windows in the properties of the installed query and the properties of the new query. In this case, the join only needs to be computed once and the join result can be shared for both queries. However, different selection predicates and projections might be applied to the shared join result to obtain the exact result for each query. We demonstrate this case using $q_5$ of Figure 10(a) as the query already installed and $q_6$ as the newly arriving query. Figure 13(a) illustrates the evaluation of both queries. The figure uses subscripts to distinguish equally named elements from different input streams. The upper part of the figure depicts the resulting APF *after* matching and merging the APFs of the two queries. The window definitions stay the same since both queries use the same windows. The join annotation reflects the relaxed join condition which is equal to the join condition of $q_6$ in this case. This is due to the fact that the join condition of $q_5$ implies the join condition of $q_6$. The lower part of the figure shows the application of further selection and projection operators to generate the final results for queries $q_5$ and $q_6$. Since both queries use the same window definitions for their corresponding input streams, the basic join is computed only once as result of the widening. The join result is then further processed using according selection and projection operators in the postprocessing phase to obtain the final results for both queries.

**Selective join result sharing** Similar to full join result sharing, selective join result sharing also allows to compute the join result once and to share it for both queries. This case occurs if $\Delta \neq \Delta'$ for at least one pair and $\mu = \mu'$ for each pair of corresponding data windows in the properties of the installed query and the properties of the new query. Apart from the selection and projection operators as in full join result sharing, an additional selection of join results is necessary in the selection phase. The reason is that during widening, the window size of each data window that has non-equal size in the properties of the installed query and the properties of the new query is set to the maximum of the corresponding window sizes in both properties. Figure 13(b) illustrates this aspect using queries $q_5$ and $q_7$ as an example. Query $q_5$ defines a window size of 10 for the input stream `photon1` and a window size of 20 for the input stream `photon2`. Accordingly, query $q_7$ defines a window size of 30 for the input stream `photon1` and a window size of 15 for the input stream `photon2`. Consequently, the shareable window size for stream `photon1` is $\max(10, 30) = 30$ and the shareable window size for stream `photon2` is $\max(20, 15) = 20$. In the selection phase, the operator SelectJoinResult(10,20) selects only those join result items where the joined item from the left input is within the first 10 units of the widenend data window of the left input stream, i. e., it selects only the first third of the entire widened window which has a total size of 30 units. Furthermore, the joined item from the right input of each selected join result item is situated within the first 20 units of the widened data window of the right input stream, i. e., the operator selects the entire widened window which has a total size of 20 units to obtain the correct results for $q_5$. Units may either be time units or the number of elements, depending on whether time-based or count-based windows are used. The situation is similar for the operator SelectJoinResult(30,15) which selects the entire widenend window of the left input stream and only the first three quarters of the widened window of the right input stream to obtain the correct results for $q_7$. Again, the final results for both queries are obtained by applying adequate selection and projection operators in the postprocessing phase.

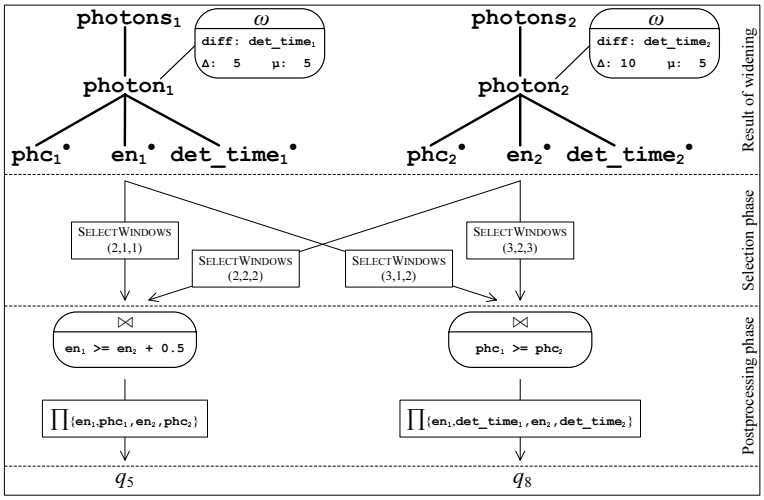
The SelectJoinResult operator requires additional information for each join result item during the selection phase. For time-based data windows, the timestamps of the individual input items forming a join result item need to be preserved if the join query removes the timestamp elements from the join result. For count-based data windows, we associate each individual input item with a monotonically increasing integer value when the item enters the corresponding data window. Furthermore, each input item is associated with the lower bound of the corresponding data window instance the input item belonged to when the respective join result item was generated. Join computations are triggered by window updates in our join semantics. Therefore, the corresponding window instance of the join input stream that triggered the join computation corresponds to the window instance of the updated window. Window bounds are time values in case of time-based data windows and counter values in case of count-based data windows. Knowing the window lower bounds and the timestamp or counter values of the joined data items enables




**(a)** Full join result sharing between $q_5$ and $q_6$




**(b)** Selective join result sharing between $q_5$ and $q_7$




**(c)** Selective window sharing between $q_5$ and $q_8$

**Figure 13:** Join result sharing

us to decide which join result items qualify for the result of a certain join query. Note that the counter value for count-based windows does not need to grow indefinitely. It can be reset during any window update process by subtracting the minimum of the counter values of all the data items contained in the window from the current window bounds and from the counter values of all the items remaining in the window after the update. Newly added items subsequently need to be consistently associated with further incremental counter values.

**Selective window sharing** The third and final case is called selective window sharing and occurs if $\mu \neq \mu'$ for at least one pair of corresponding data windows in the properties of the installed query and the properties of the new query. In this case, the join result cannot be shared due to the incompatible window definitions. Instead, we can compute relaxed window definitions that are shareable by both queries, just as we do for aggregate queries. Figure 13(c) shows an example using queries $q_5$ and $q_8$. Widening computes the new window definitions for both input streams and removes the join annotation. In the selection phase, a window selection operator selects the appropriate windows in the appropriate order to generate windows of the window size and the step size required by the respective query. For example, in Figure 13(c), SelectWindows(2,1,1) selects two consecutive windows of window size 5 and step size 5 and combines them to generate a window of window size 10 and step size 5 corresponding to the window over the left input of $q_5$. SelectWindows(2,2,2) analogously combines two windows of window size 10 and step size 5 to generate the window over the right input of $q_5$ with window size 20 and step size 10. But it only takes into account every second window in the input when generating a particular window. This provides for the combination of contiguous non-overlapping windows. Also, only after every second window arriving on the input stream, the window of $q_5$ is updated. This leads to the required step size of 10 whereas the shared windows have a step size of 5. Similarly, for $q_8$, SelectWindows(3,1,2) selects three consecutive windows of window size 5 and step size 5 to form one window of window size 15 over the left input of $q_8$. Only after every second window arriving on the input stream, the window of $q_8$ is updated. This leads to the required step size of 10, which is two times the step size of the shared window. Finally, SelectWindows(3,2,3) combines three windows of window size 10 to form one window of window size 30 over the right input of $q_8$. Only every second input window is used to get a sequence of contiguous non-overlapping windows for a particular window instance. To obtain the correct window update interval of 15 for the window over the right input of $q_8$, three windows of step size 5 must have arrived on the shared input before updating the window of $q_8$. The postprocessing phase then generates the final join result for both queries by applying appropriate join and projection operators.

The joins in the postprocessing phase obey the join semantics introduced in Section 5.1.1. The join operators can derive the updated parts of the data windows delivered by the SelectWindows operators by means of the window definition, i. e., by examining the window bounds and the step size. If the SelectWindows operators and the join operators are kept separate as indicated in Figure 13(c), overlapping parts of subsequent data windows are delivered to the join operators multiple times. This can be avoided by integrating the SelectWindows operators of the selection phase with the join operators of the postprocessing phase and by applying appropriate optimizations.

Finally, it is worth noting that it might be more efficient in practice to execute each window join operator individually on the ungrouped inputs instead of computing and sharing common windows among queries via selective window sharing. Deciding which solution is the better choice depends on cost function and network characteristics.

# 6 Evaluation

To assess the benefits of data stream widening, we have conducted some performance experiments using our StreamGlobe prototype implementation. We have implemented data stream sharing and data stream widening in StreamGlobe, together with a naive strategy called data shipping that merely serves as a baseline. For each query, data shipping individually transmits each original input stream referenced in the query from the peer where the corresponding original stream is registered to the peer that registered the query. The transmission uses a shortest path in the network without sharing or forking the stream in any way. We have implemented StreamGlobe and all optimization strategies using Java 6 and ran our tests on a blade server. Depending on the scenario, we used 8 or 16 blades, one for each peer in the

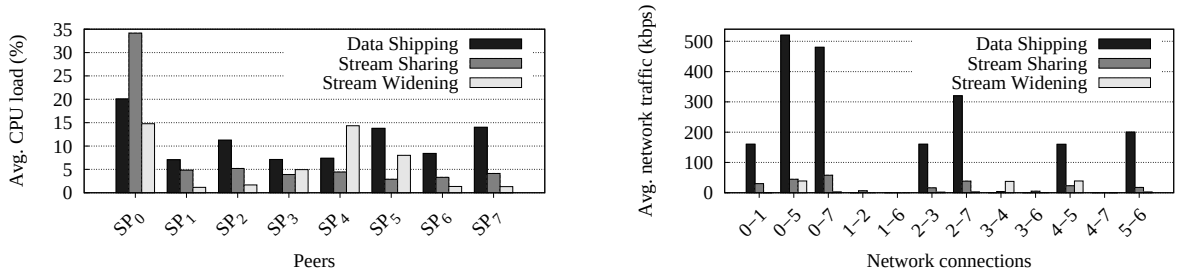



**Figure 14:** Average CPU load and network traffic

| | CPU Load (%) | | |
|---|---|---|---|
| | Accumulated | Average | Percentage |
| Data Shipping | 89.0 | 11.13 | 100.0% |
| Stream Sharing | 62.8 | 7.85 | 70.6% |
| Stream Widening | 47.6 | 5.95 | 53.5% |
| | Network Traffic (kbps) | | |
| | Accumulated | Average | Percentage |
| Data Shipping | 1999.7 | 166.6 | 100.0% |
| Stream Sharing | 243.6 | 20.3 | 12.2% |
| Stream Widening | 127.8 | 10.6 | 6.4% |

**Table 1:** Accumulated and average overall CPU load and network traffic

backbone network. Each blade ran CentOS 4 and was equipped with a 2.8 GHz Intel Xeon processor and at least 1 GB of main memory.

We conducted performance tests using various scenarios differing in the number of peers in the backbone network (8 or 16) and in the number of queries registered (from 4 to 100). We used three- and four-dimensional hypercubes as network topologies. The data streams were of the form described in Section 1. Since results for all scenarios were similar, we selected one for presentation. The chosen scenario uses a three-dimensional hypercube network topology as shown in Figure 1. We used a single `photons` data stream and registered 32 randomly generated queries. The query generator generates queries that return a randomly chosen subset of the elements contained in the input streams. It further generates random selection predicates. In our case, selections were performed either on the detector pixel coordinates (`dx`, `dy`) or on the energy (`en`) of a photon, or on both. Selections consist of conjunctive and disjunctive combinations of atomic predicates. An atomic predicate in turn consists of an element variable, a comparison operator, and a constant, e. g., `en >= 1.3`. The constants are chosen randomly from a predefined set of reasonable values from our `photons` data set using a normal distribution. This already allows for some amount of sharing even without data stream widening.

Figure 14 shows the results in terms of average CPU load in percent on the peers in the example network and in terms of average network traffic in kilobits per second on the network connections between peers. Additionally, Table 1 presents the accumulated and average CPU load and network traffic in the overall backbone network. The accumulated overall values are computed by adding up the average CPU load and network traffic values shown in Figure 14 for all peers and network connections. The average overall values are computed by dividing the accumulated values by the number of peers or network connections, respectively. The percentage in the table illustrates the relation between the three strategies compared to the values of data shipping which serve as the baseline at 100%.

As expected, data shipping causes the highest amount of CPU load and network traffic throughout the network since it requires to forward the entire data stream multiple times, once for each query. In contrast, stream sharing potentially shares one result data stream for satisfying multiple subscriptions. Thus, it reduces computational load and network traffic due to result sharing. Also, by installing subscriptions close to the data sources in the network, early filtering and early aggregation at the stream source further reduce resource usage within the backbone network. Only at $SP_0$, which is the peer where the original

`photons` data stream is registered, the CPU load increases using stream sharing since queries that are unable to share any preprocessed streams in the network are installed at the stream source and their results are routed to the querying peer on a shortest path in the network. In our scenario, 14 of the 32 queries registered were able to share preprocessed streams without widening. Using data stream widening, this value increased to 31, i. e., every query except for the first one was able to reuse a possibly widened result data stream of a previously registered query. This obviously leads to a further reduction of CPU load and network traffic. In the scenario, the average CPU load in the overall network, i. e., averaged over all 8 peers, dropped from 7.85% to 5.95%, a reduction of about 25%. The average network traffic in the overall network, i. e., averaged over all 12 network connections, dropped from 20.3 kbps to 10.6 kbps, a reduction of about 48%.

The results show that data stream widening serves the important purpose of making optimization quality more independent of the actual query characteristics and the query registration sequence. Thus, data stream widening achieves good optimization results and enables efficient resource usage for arbitrary query loads.

Due to the increased optimization overhead, registering a query usually takes longer when using stream widening compared to mere stream sharing without widening. Query registration times tend to be longer for both strategies the more queries have already been registered in the system. This is due to the fact that the optimizer has more alternatives that it can take into account. While stream widening caused an increase in query registration times of up to double the amount of time used by stream sharing without widening, registering a query never took longer than 45 seconds in the largest scenario with 100 queries registered. Since we deal with continuous queries which are supposed to run for several hours, days, weeks, or even months, optimization delays of several seconds up to some minutes for a single query are acceptable. Further, we may stop the optimization process after a certain amount of time and use the best solution found so far if query registration times should not exceed a certain threshold.

# 7 Related Work

The main application domain for the techniques presented in this paper is the optimization of resource usage in a distributed DSMS. Numerous DSMSs have been proposed in recent years. Among them is STREAM [3], which uses the Continuous Query Language (CQL) [4] for registering subscriptions. CQL is based on SQL and introduces an additional syntax for the specification of data windows over streams just as WXQuery does with respect to XQuery. STREAM processes data streams by transforming streams into relations and by transforming the query results back into streams again. In contrast, StreamGlobe processes XML data streams directly using an augmented fragment of the XQuery language. TelegraphCQ [11] adaptively processes data streams using, among other things, the Eddy [8] approach for adaptive tuple routing. NiagaraCQ [13] optimizes query processing by sharing common computations among continuous queries through appropriately grouping queries according to similar structures. This is related to our more powerful approach that allows for the dynamic adaptation of streams. PIPES [24] takes a different approach by providing a public infrastructure offering essential building blocks for developing DSMSs.

All of the above systems are centralized and tuple-based whereas StreamGlobe constitutes a distributed DSMS for managing XML data streams. Aurora [10] is another centralized data flow system that processes tuple streams. With Aurora* and Medusa [14], a decentralized version of Aurora and a distributed infrastructure supporting federated operation of nodes also exist. Further development aiming at enabling new DSMS functionality such as dynamic revision of query results, dynamic query modification, and flexible optimization led to the Borealis system [1].

Data stream sharing is closely related to multi-query optimization (MQO) [38, 39]. Traditional MQO mainly aims at optimizing the evaluation of a query batch over a set of persistent data. However, the streaming paradigm opens many new possibilities in our setting compared to traditional MQO, which is mainly due to the dynamic nature of streaming data and the persistent nature of continuous queries over data streams. For example, we can dynamically widen data streams by relaxing predicates or window definitions to make an initially unsuitable stream shareable. Furthermore, it is possible to narrow a data stream if some of its data is not needed any more due to the deletion of queries.

Further, the problem of query containment has a strong relation to data stream sharing. Query

containment has already been studied for querying XML data, mainly in the context of optimizing query rewriting in peer data management systems (PDMSs) [43]. This also includes dealing with nested queries [16]. A main application area of query containment is semantic caching [15]. However, as with MQO, the main difference to our work lies in the fact that, instead of dealing with persistent data and volatile queries, we are dealing with persistent queries and volatile data. Therefore, in our setting, the cached data corresponds to the—albeit volatile—data flowing through the network.

Another related subject are XML views. Among other things, efficiently supporting queries over XML views of relational data for increased flexibility and interoperability is a major issue in this context. The IBM XML Query Graph Model (XQGM) [40] is a graph-based internal query representation for XQueries over XML views of relational data used in the XPERANTO middleware system. An incoming XQuery is directly translated into an XQGM by the query parser and the internal query representation is used to employ query rewriting optimizations and to compose the query with the views it references. The XQGM is subsequently processed and decomposed into two parts. One part captures the memory and data intensive processing and is pushed down to the relational engine while the other part constitutes a tagger graph structure used to construct the XML query result. This approach is clearly related to ours since we also use an abstract internal query representation for optimizing (W)XQuery processing. The main differences are that StreamGlobe does not use a relational backend but directly processes XML data and that we deal with data stream processing in a distributed environment. While the XQGM approach mainly aims at exploiting the facilities of proven relational database backends for efficiently processing XQueries over a flexible and interoperable XML view interface, StreamGlobe targets efficient resource usage in a distributed DSMS by means of sharing common work and data among multiple long-running continuous queries.

The importance of sharing work and resources to achieve efficient and scalable query processing has been observed multiple times in the literature, especially in the context of data streams. One approach in this direction is to enable shared computation for multiple related aggregations over data streams that differ only in the choice of grouping attributes [50]. Precise sharing of common work while avoiding unnecessary work is the focus of TULIP [27], which uses the concept of tuple lineage known, e. g., from Eddies[8], to keep track of predicate evaluation results. Resource sharing for continuous sliding window aggregates is an aspect that has sparked special interest. Various algorithms for solving this issue have been proposed [6]. A possible approach for enabling sharing for overlapping sliding windows is to divide the windows into disjoint *panes* [32]. These can be used to compute window aggregates containing the respective panes. Therefore, work on the overlapping parts of the windows is done only once. An improvement over panes allows sharing among queries involving different window definitions *and* selection predicates at the same time using so-called *shards* [28]. Further work shows how to enable multi-query optimization for sliding window aggregates by means of schedule synchronization [17] or focuses on general semantics and evaluation techniques for window aggregates over data streams [33].

Joins over data streams are widely covered in the literature. As in our work, most approaches use windows to limit the memory requirements of stream-based joins. Many of these solutions use join semantics similar to the traditional window join semantics described in Section 5.1.1. Some approaches, e. g. CACQ [34], also use a basic approach for sharing join results among multiple queries with different window definitions. This involves computing the join of the contents of the largest windows and then filtering the result multiple times with different filter conditions to obtain the exact results for all queries. However, this might impose considerable delay on queries using relatively small windows since these queries have to wait for their results until the join of the larger windows completes. Alternative algorithms for shared window join scheduling [19] alleviate this problem. The literature furthermore provides efficient algorithms for processing sliding window multi-joins in continuous queries over data streams [18]. A new paradigm of multi-query optimization for window queries over data streams suggests the slicing of window states and introduces a new pipelining method to reduce the number of total joins [48].

# 8 Conclusion

In this paper, we have introduced an abstract property tree (APT) for representing, matching, and merging queries and data in a distributed DSMS. The presented approach enables data stream sharing as well as data stream widening and data stream narrowing. We have established formal rules for the transla-

tion of a query formulated in our XQuery-based subscription language WXQuery into a corresponding APT. Query templates provide for the inverse translation. Further, we have extended our approach to support queries with multiple inputs, e. g., join queries, by introducing abstract property forests (APFs). The results of performance experiments conducted using the prototype implementation of our distributed DSMS StreamGlobe demonstrate the effectiveness of data stream sharing in combination with data stream widening at a reasonable optimization cost.

An interesting topic for future work is the investigation of the cost-efficient placement of join operators in the StreamGlobe network. Techniques from distributed databases may be useful in this direction. Furthermore, the problem of dynamic plan migration, i. e., of replacing a query evaluation plan with its widened or narrowed pendant in the network without losing data, is of great importance. Examining previously proposed solutions to this problem [51, 26, 49] with regard to their applicability in our setting can give directions on how to solve this issue. Additional difficulties for dynamic plan migration in the context of data stream widening in StreamGlobe arise from the fact that other queries may depend on an existing plan. These dependencies must be preserved during plan migration. A further interesting aspect is the extension of the WXQuery subscription language, e. g., by introducing a general `let` expression similar to that of standard XQuery. Finally, the tree algebra introduced in this chapter can be extended, e. g., to support tree subtraction. Among other things, tree subtraction would be a useful approach for computing remainder queries in semantic caching.

# References


[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, pages 277–289, Asilomar, CA, USA, Jan. 2005.

[2] Y. Ahmad and U. Çetintemel. Network-Aware Query Processing for Stream-based Applications. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 456–467, Toronto, Canada, Aug. 2004.

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, Mar. 2003.

[4] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *Proc. of the Int'l Workshop on Database Programming Languages (DBPL)*, pages 1–19, Potsdam, Germany, Sept. 2003.

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

[6] A. Arasu and J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 336–347, Toronto, Canada, Aug. 2004.

[7] B. Aschenbach. Discovery of a young nearby supernova remnant. *Nature*, 396(6707):141–142, Nov. 1998.

[8] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 261–272, Dallas, TX, USA, May 2000.

[9] I. Botan, P. M. Fischer, D. Florescu, D. Kossmann, T. Kraska, and R. Tamosevicius. Extending XQuery with Window Functions. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 75–86, Vienna, Austria, Sept. 2007.

[10] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams – A New Class of Data Management Applications. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 215–226, Hong Kong, China, Aug. 2002.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, Jan. 2003.

[12] S. Chandrasekaran and M. J. Franklin. Streaming Queries over Streaming Data. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 203–214, Hong Kong, China, Aug. 2002.

[13] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 379–390, Dallas, TX, USA, May 2000.

[14] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, Jan. 2003.

[15] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 330–341, Mumbai (Bombay), India, Sept. 1996.

[16] X. Dong, A. Y. Halevy, and I. Tatarinov. Containment of Nested XML Queries. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 132–143, Toronto, Canada, Aug. 2004.

[17] L. Golab, K. G. Bijay, and M. T. Özsu. Multi-Query Optimization of Sliding Window Aggregates by Schedule Synchronization. In *Proc. of the ACM Int'l Conf. on Information and Knowledge Management (CIKM)*, pages 844–845, Arlington, VA, USA, Nov. 2006.

[18] L. Golab and M. T. Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 500–511, Berlin, Germany, Sept. 2003.

[19] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 297–308, Berlin, Germany, Sept. 2003.

[20] C. Heinrich. *RFID and Beyond: Growing Your Business through Real World Awareness*. Wiley & Sons, 2005.

[21] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating Window Joins over Unbounded Streams. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, pages 341–352, Bangalore, India, Mar. 2003.

[22] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 1309–1312, Toronto, Canada, Aug. 2004.

[23] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 228–239, Toronto, Canada, Aug. 2004.

[24] J. Krämer and B. Seeger. PIPES – A Public Infrastructure for Processing and Exploring Streams. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 925–926, Paris, France, June 2004.

[25] J. Krämer and B. Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *Proc. of the Int'l Conf. on Management of Data (COMAD)*, pages 70–82, Goa, India, Jan. 2005.

[26] J. Krämer, Y. Yang, M. Cammert, B. Seeger, and D. Papadias. Dynamic Plan Migration for Snapshot-Equivalent Continuous Queries in Data Stream Systems. In *Proc. of the Int'l Conf. on Semantics of a Networked World (ICSNW)*, pages 497–516, Munich, Germany, Mar. 2006.

[27] S. Krishnamurthy, M. J. Franklin, J. M. Hellerstein, and G. Jacobson. The Case for Precision Sharing. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 972–983, Toronto, Canada, Aug. 2004.

[28] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-Fly Sharing for Streamed Aggregation. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 623–634, Chicago, IL, USA, June 2006.

[29] R. Kuntschke and A. Kemper. Data Stream Sharing. In *Proc. of the Int'l Workshop on Pervasive Information Management (PIM)*, pages 45–56, Munich, Germany, Mar. 2006.

[30] R. Kuntschke and A. Kemper. Matching and Evaluation of Disjunctive Predicates for Data Stream Sharing. In *Proc. of the ACM Int'l Conf. on Information and Knowledge Management (CIKM)*, pages 832–833, Arlington, VA, USA, Nov. 2006.

[31] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *Proc. of the ACM SIGACT–SIGMOD–SIGART Symp. on Principles of Database Systems (PODS)*, pages 95–104, San José, CA, USA, May 1995.

[32] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. *ACM SIGMOD Record*, 34(1):39–44, Mar. 2005.

[33] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 311–322, Baltimore, MD, USA, June 2005.

[34] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries over Streams. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 49–60, Madison, WI, USA, June 2002.

[35] A. Marian and J. Siméon. Projecting XML Documents. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 213–224, Berlin, Germany, Sept. 2003.

[36] A. Marian and J. Siméon. Projecting XML Documents. Technical Report, Columbia University, Feb. 2003.

[37] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, page 49, Atlanta, GA, USA, Apr. 2006.

[38] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 249–260, Dallas, TX, USA, May 2000.

[39] T. K. Sellis. Multiple-Query Optimization. *ACM Trans. on Database Systems (TODS)*, 13(1):23–52, Mar. 1988.

[40] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 261–270, Roma, Italy, Sept. 2001.

[41] U. Srivastava, K. Munagala, and J. Widom. Operator Placement for In-Network Stream Query Processing. In *Proc. of the ACM SIGACT–SIGMOD–SIGART Symp. on Principles of Database Systems (PODS)*, pages 250–258, Baltimore, MD, USA, June 2005.

[42] B. Stegmaier, R. Kuntschke, and A. Kemper. StreamGlobe: Adaptive Query Processing and Optimization in Streaming P2P Environments. In *Proc. of the Int'l Workshop on Data Management for Sensor Networks (DMSN)*, pages 88–97, Toronto, Canada, Aug. 2004.

[43] I. Tatarinov and A. Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 539–550, Paris, France, June 2004.

[44] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 15(3):555–568, May 2003.

[45] W. Voges, B. Aschenbach, T. Boller, H. Bräuninger, U. Briel, W. Burkert, K. Dennerl, J. Englhauser, R. Gruber, F. Haberl, G. Hartner, G. Hasinger, M. Kürster, E. Pfeffermann, W. Pietsch, P. Predehl, C. Rosso, J. H. M. M. Schmitt, J. Trümper, and H. U. Zimmermann. The ROSAT All-Sky Survey Bright Source Catalogue. *Astronomy and Astrophysics*, 349(2):389–405, July 1999.

[46] W3C. XQuery 1.0: An XML Query Language (W3C Recommendation, January 23rd, 2007), Jan. 2007. `http://www.w3.org/TR/xquery/`.

[47] W3C. XQuery 1.0 and XPath 2.0 Formal Semantics (W3C Recommendation, January 23rd, 2007), Jan. 2007. `http://www.w3.org/TR/xquery-semantics/`.

[48] S. Wang, E. Rundensteiner, S. Ganguly, and S. Bhatnagar. State-Slice: New Paradigm of Multi-query Optimization of Window-based Stream Queries. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 619–630, Seoul, Korea, Sept. 2006.

[49] Y. Yang, J. Krämer, D. Papadias, and B. Seeger. HybMig: A Hybrid Approach to Dynamic Plan Migration for Continuous Queries. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 19(3):398–411, Mar. 2007.

[50] R. Zhang, N. Koudas, B. C. Ooi, and D. Srivastava. Multiple Aggregations Over Data Streams. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 299–310, Baltimore, MD, USA, June 2005.

[51] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 431–442, Paris, France, June 2004.

# A Inference Rules for Translating a WXQuery into an APF

This section extends the formal rules for translating a WXQuery into a corresponding APT introduced in Section 3.2.4 to additionally support join queries and their translation into APFs.

The construction of the APTs of each individual input stream works as described in Section 3.2.4. For queries with multiple input data streams, instead of generating a single APT, we generate a list of APTs containing one APT per input data stream. The length of the list can be derived from the query in advance. For each input source $i$, we also determine the input stream identifier or document name $id^i$ and the corresponding DTD $d^i$ in advance during a preprocessing phase by scanning the query for any `stream` or `doc` function calls which contain an input source identifier as parameter. We use the input source identifier to retrieve the corresponding DTD from a metadata repository. Therefore, $id^i$ and $d^i$ are already present for each input source $i$ and the following rules simply forward them. This is important since we need the corresponding input source identifier during APF generation to add paths, annotations, and output elements to the correct APTs. In this context, we also slightly redefine the semantics of the $\mathsf{path}$, $\overline{\mathsf{path}}$, and $\mathsf{cond}$ functions. These now only return those paths or conditions referring to the corresponding input source as indicated by a superscript. For example, if $Path_1$ represents a path belonging to APT $t^i$, then $\overline{\mathsf{path}}^i(Path_1)$ returns this path and $P^i \cup \{\overline{\mathsf{path}}^i(Path_1)\}$ adds it to $P^i$. At the same time, $\overline{\mathsf{path}}^j(Path_1)$ does not return any path and therefore $P^j \cup \{\overline{\mathsf{path}}^j(Path_1)\}$ does not add any path to $P^j$ for all $1 \leq j \leq m, j \neq i$. Finally, we define the $\mathsf{cond}^i$ function to only return conditions that exclusively reference elements from input source $i$, i. e., simple selection conditions. Further, we additionally introduce a variant $\overline{\mathsf{cond}}^i$ that only returns conditions that reference additional input sources besides $i$, i. e., join conditions involving input source $i$.

We again use the inference rule notation of the XQuery formal semantics specification [47]. The judgment

$$Env \vdash \alpha \Rightarrow [(P^1, A^1, O^1, id^1, d^1), \ldots, (P^m, A^m, O^m, id^m, d^m)]$$

holds if and only if, under the environment $Env$, the expression $\alpha$ induces the construction of the APTs $(P^1, A^1, O^1, id^1, d^1)$ to $(P^m, A^m, O^m, id^m, d^m)$ as described in Section 3.2.4. Inference rules are again of the form

$$\frac{premise_1 \ \ldots \ premise_n}{conclusion}$$

where all premises and the conclusion are judgments of the above form. Additionally, premises may again constitute expressions of the form $Env' = Env + (\$var \Rightarrow Path)$ that extend the environment $Env$ yielding the environment $Env'$ by adding the binding of the variable $\$var$ to the path represented by $Path$. The inference rule expresses that, if all premises hold, then the conclusion holds as well.

We now give the extended inference rules for each WXQuery expression of Definition 2.1.

**Empty direct element constructor** An empty direct element constructor induces a list of empty APTs.

$$\frac{}{Env \vdash \texttt{<t/>} \Rightarrow [(\emptyset^1, \emptyset^1, \emptyset^1, id^1, d^1), \ldots, (\emptyset^m, \emptyset^m, \emptyset^m, id^m, d^m)]} \tag{11}$$

**Direct element constructor** For each input source, the rule generates the corresponding APT just as for the single APT in the original rule of Section 3.2.4.

$$\frac{\begin{array}{c} Env \vdash \alpha_1 \Rightarrow [(P_1^1, A_1^1, O_1^1, id^1, d^1), \ldots, (P_1^m, A_1^m, O_1^m, id^m, d^m)] \\ \ldots \\ Env \vdash \alpha_n \Rightarrow [(P_n^1, A_n^1, O_n^1, id^1, d^1), \ldots, (P_n^m, A_n^m, O_n^m, id^m, d^m)] \end{array}}{\begin{array}{c} Env \vdash \texttt{<t>}\alpha_1 \ldots \alpha_n\texttt{</t>} \\ \Rightarrow [(\bigcup_{i=1}^n P_i^1, \bigcup_{i=1}^n A_i^1, \bigcup_{i=1}^n O_i^1, id^1, d^1), \\ \ldots, \\ (\bigcup_{i=1}^n P_i^m, \bigcup_{i=1}^n A_i^m, \bigcup_{i=1}^n O_i^m, id^m, d^m)] \end{array}} \tag{12}$$

Note that, as described in Section 3.2.4, we have again rephrased the WXQuery expression for direct element constructors in the inference rule compared to the WXQuery definition to better support the inference rule notation.

**FLWR expression** We again split the inference rule for FLWR expressions into four separate rules as in Section 3.2.4. We use the same shortcuts and functions as in the introduction of the original rules. Additionally, as introduced above, the functions $\mathsf{path}^i$ and $\overline{\mathsf{path}}^i$ with $1 \leq i \leq m$ return only those paths that reference the input source with identifier $id^i$. Furthermore, the $\mathsf{cond}$ function only considers non-join conditions, i. e., conditions that reference elements from only one APT. We introduce the $\overline{\mathsf{cond}}$ function to exclusively handle join conditions referencing elements from more than one APT. Both functions are applied as described in Section 3.2.4 on the respective conditions.

First, we consider a `for` loop without any data window. The first premise in the rule again reflects the variable binding in the `for` loop.

$$\frac{\begin{array}{c} Env' = Env + (\$x \Rightarrow \mathsf{path}(Path_1)) \\ Env' \vdash \alpha \Rightarrow [(P^1, A^1, O^1, id^1, d^1), \ldots, (P^m, A^m, O^m, id^m, d^m)] \end{array}}{\begin{array}{c} Env \vdash \texttt{for}\ \$x\ \texttt{in}\ Path_1\ \texttt{where}\ \chi\ \texttt{return}\ \alpha \\ \Rightarrow [(P^1 \cup \overline{\mathsf{path}}^1(Path_1) \cup \overline{\mathsf{path}}^1(\chi), \\ A^1 \cup \{(\sigma, \mathsf{cond}^1(Path_1) \cup \mathsf{cond}^1(\chi), O^1), \\ (\bowtie, \overline{\mathsf{cond}}^1(Path_1) \cup \overline{\mathsf{cond}}^1(\chi), O^1)\}, O^1, id^1, d^1), \\ \ldots, \\ (P^m \cup \overline{\mathsf{path}}^m(Path_1) \cup \overline{\mathsf{path}}^m(\chi), \\ A^m \cup \{(\sigma, \mathsf{cond}^m(Path_1) \cup \mathsf{cond}^m(\chi), O^m), \\ (\bowtie, \overline{\mathsf{cond}}^m(Path_1) \cup \overline{\mathsf{cond}}^m(\chi), O^m)\}, O^m, id^m, d^m)] \end{array}} \tag{13}$$

Note that an annotation is only added to the set $A^i$ of annotations of a certain APT $t^i$ if $O^i$ is not empty, i. e., the annotation is associated with at least one returned element in $O^i$. As stated in Section 2, $O := O^1 \cup \cdots \cup O^m$ must not be empty, i. e., each query must return at least one element of the input sources or an aggregate value based on the input sources. Therefore, each annotation is associated with at least one element in at least one of the APTs of the APF.

The next rule describes the translation of a `for` loop with a count-based data window. The selection annotations are again optional, just as the corresponding conditions in the query.

$$\frac{\begin{array}{c} Env' = Env + (\$x \Rightarrow \mathsf{path}(Path_1)) \\ Env' \vdash \alpha \Rightarrow [(P^1, A^1, O^1, id^1, d^1), \ldots, (P^m, A^m, O^m, id^m, d^m)] \end{array}}{\begin{array}{c} Env \vdash \texttt{for}\ \$x\ \texttt{in}\ Path_1\ \texttt{|count}\ \Delta\ \texttt{step}\ \mu\texttt{|}\ \texttt{where}\ \chi\ \texttt{return}\ \alpha \\ \Rightarrow [(P^1 \cup \overline{\mathsf{path}}^1(Path_1) \cup \overline{\mathsf{path}}^1(\chi), \\ A^1 \cup \{(\omega, (\text{count}, \Delta, \mu), \mathsf{path}^1(Path_1)), (\text{pre-}\sigma, \mathsf{cond}^1(Path_1), \omega), \\ (\text{post-}\sigma, \mathsf{cond}^1(\chi), \omega), (\bowtie, \overline{\mathsf{cond}}^1(Path_1) \cup \overline{\mathsf{cond}}^1(\chi), O^1)\}, O^1, id^1, d^1), \\ \ldots, \\ (P^m \cup \overline{\mathsf{path}}^m(Path_1) \cup \overline{\mathsf{path}}^m(\chi), \\ A^m \cup \{(\omega, (\text{count}, \Delta, \mu), \mathsf{path}^m(Path_1)), (\text{pre-}\sigma, \mathsf{cond}^m(Path_1), \omega), \\ (\text{post-}\sigma, \mathsf{cond}^m(\chi), \omega), (\bowtie, \overline{\mathsf{cond}}^m(Path_1) \cup \overline{\mathsf{cond}}^m(\chi), O^m)\}, O^m, id^m, d^m)] \end{array}} \tag{14}$$

Note that $\mathsf{path}^i(Path_1)$ only returns a path for the input source with identifier $id^i$. Therefore, the rule generates the window annotation only once and associates it with the correct input source. The rule does not generate the same window annotation for the other input sources since for the other sources, the parent of the annotation specified by $\mathsf{path}^j(Path_1)$ with $j \neq i$ is empty.

The inference rule describing the translation of `for` loops with time-based data windows again handles an additional path $Path_2$ which identifies the window reference element.

$$\frac{\begin{array}{c} Env' = Env + (\$x \Rightarrow \mathsf{path}(Path_1)) \\ Env' \vdash \alpha \Rightarrow [(P^1, A^1, O^1, id^1, d^1), \ldots, (P^m, A^m, O^m, id^m, d^m)] \end{array}}{\begin{array}{c} Env \vdash \texttt{for}\ \$x\ \texttt{in}\ Path_1\ \texttt{|}\ Path_2\ \texttt{diff}\ \Delta\ \texttt{step}\ \mu\texttt{|}\ \texttt{where}\ \chi\ \texttt{return}\ \alpha \\ \Rightarrow [(P^1 \cup \overline{\mathsf{path}}^1(Path_1) \cup \overline{\mathsf{path}}^1(Path_2) \cup \overline{\mathsf{path}}^1(\chi), \\ A^1 \cup \{(\omega, (\text{diff}, \mathsf{path}^1(Path_2), \Delta, \mu), \mathsf{path}^1(Path_1)), \\ (\text{pre-}\sigma, \mathsf{cond}^1(Path_1), \omega), (\text{post-}\sigma, \mathsf{cond}^1(\chi), \omega), \\ (\bowtie, \overline{\mathsf{cond}}^1(Path_1) \cup \overline{\mathsf{cond}}^1(\chi), O^1)\}, O^1, id^1, d^1), \\ \ldots, \\ (P^m \cup \overline{\mathsf{path}}^m(Path_1) \cup \overline{\mathsf{path}}^m(Path_2) \cup \overline{\mathsf{path}}^m(\chi), \\ A^m \cup \{(\omega, (\text{diff}, \mathsf{path}^m(Path_2), \Delta, \mu), \mathsf{path}^m(Path_1)), \\ (\text{pre-}\sigma, \mathsf{cond}^m(Path_1), \omega), (\text{post-}\sigma, \mathsf{cond}^m(\chi), \omega), \\ (\bowtie, \overline{\mathsf{cond}}^m(Path_1) \cup \overline{\mathsf{cond}}^m(\chi), O^m)\}, O^m, id^m, d^m)] \end{array}} \tag{15}$$

Although we do not deal with queries mixing aggregates and joins due to the semantic differences described in Section 5.1.2, we introduce the inference rule for translating `let` expressions which are used to bind the result of an aggregate function call to a variable in WXQuery. This sets the stage for supporting mixed queries in future work.

$$\frac{\begin{array}{c} Env' = Env + (\$a \Rightarrow \Phi\texttt{(}\mathsf{path}(Path_3)\texttt{)}) \\ Env' \vdash \alpha \Rightarrow [(P^1, A^1, O^1, id^1, d^1), \ldots, (P^m, A^m, O^m, id^m, d^m)] \end{array}}{\begin{array}{c} Env \vdash \texttt{let}\ \$a\ \texttt{:=}\ \Phi\texttt{(}Path_3\texttt{)}\ \texttt{where}\ \chi\ \texttt{return}\ \alpha \\ \Rightarrow [(P^1 \cup \overline{\mathsf{path}}^1(Path_3) \cup \overline{\mathsf{path}}^1(\chi), \\ A^1 \cup \{(\gamma, \Phi, \mathsf{path}^1(Path_3)), (\text{pre-}\sigma, \mathsf{cond}^1(Path_3), \gamma), (\sigma, \mathsf{cond}^1(\chi), O^1), \\ (\bowtie, \overline{\mathsf{cond}}^1(Path_3) \cup \overline{\mathsf{cond}}^1(\chi), O^1)\}, O^1, id^1, d^1), \\ \ldots, \\ (P^m \cup \overline{\mathsf{path}}^m(Path_3) \cup \overline{\mathsf{path}}^m(\chi), \\ A^m \cup \{(\gamma, \Phi, \mathsf{path}^m(Path_3)), (\text{pre-}\sigma, \mathsf{cond}^m(Path_3), \gamma), (\sigma, \mathsf{cond}^m(\chi), O^m), \\ (\bowtie, \overline{\mathsf{cond}}^m(Path_3) \cup \overline{\mathsf{cond}}^m(\chi), O^m)\}, O^m, id^m, d^m)] \end{array}} \tag{16}$$

Similar to the window annotations, the rule generates the aggregate annotation only once and associates it with the APT $t^i$ for which $\mathsf{path}^i(Path_3)$ actually returns a path. For the remaining input sources, the parent of the aggregate annotation remains empty and the annotation is therefore not generated.

**Conditional expression** Apart from handling multiple input streams, the rule for conditional expressions further differs from the corresponding rule in Section 3.2.4 in that it creates join annotations in addition to normal selection annotations if dictated by the query to be translated.

$$\frac{\begin{array}{c}Env \vdash \alpha_1 \Rightarrow [(P^1_{\alpha_1}, A^1_{\alpha_1}, O^1_{\alpha_1}, id^1, d^1), \ldots, (P^m_{\alpha_1}, A^m_{\alpha_1}, O^m_{\alpha_1}, id^m, d^m)] \\ Env \vdash \alpha_2 \Rightarrow [(P^1_{\alpha_2}, A^1_{\alpha_2}, O^1_{\alpha_2}, id^1, d^1), \ldots, (P^m_{\alpha_2}, A^m_{\alpha_2}, O^m_{\alpha_2}, id^m, d^m)]\end{array}}{\begin{array}{c}Env \vdash \texttt{if } \chi \texttt{ then } \alpha_1 \texttt{ else } \alpha_2 \\ \Rightarrow [(P^1_{\alpha_1} \cup P^1_{\alpha_2} \cup \overline{\mathsf{path}}^1(\chi), A^1_{\alpha_1} \cup A^1_{\alpha_2} \cup \{(\sigma, \mathsf{cond}^1(\chi), O^1_{\alpha_1}), \\ (\sigma, \mathsf{cond}^1(\neg\chi), O^1_{\alpha_2}), (\bowtie, \overline{\mathsf{cond}}^1(\chi), O^1_{\alpha_1}), (\bowtie, \overline{\mathsf{cond}}^1(\neg\chi), O^1_{\alpha_2})\}, \\ O^1_{\alpha_1} \cup O^1_{\alpha_2}, id^1, d^1), \\ \ldots, \\ (P^m_{\alpha_1} \cup P^m_{\alpha_2} \cup \overline{\mathsf{path}}^m(\chi), A^m_{\alpha_1} \cup A^m_{\alpha_2} \cup \{(\sigma, \mathsf{cond}^m(\chi), O^m_{\alpha_1}), \\ (\sigma, \mathsf{cond}^m(\neg\chi), O^m_{\alpha_2}), (\bowtie, \overline{\mathsf{cond}}^m(\chi), O^m_{\alpha_1}), (\bowtie, \overline{\mathsf{cond}}^m(\neg\chi), O^m_{\alpha_2})\}, \\ O^m_{\alpha_1} \cup O^m_{\alpha_2}, id^m, d^m)]\end{array}} \tag{17}$$

**Output of subtrees reachable from node $\$y$ through path $\pi$** In addition to the corresponding rule of Section 3.2.4, this inference rule handles multiple input streams and join annotations. In the rule, $Path_4$ again represents the pattern $\$y/\pi$.

$$\frac{}{\begin{array}{c}Env \vdash Path_4 \\ \Rightarrow [(\overline{\mathsf{path}}^1(Path_4), \\ \{(\sigma, \mathsf{cond}^1(Path_4), \{\mathsf{path}^1(Path_4)\}), (\bowtie, \overline{\mathsf{cond}}^1(Path_4), \{\mathsf{path}^1(Path_4)\})\}, \\ \{\mathsf{path}^1(Path_4)\}, id^1, d^1), \\ \ldots, \\ (\overline{\mathsf{path}}^m(Path_4), \\ \{(\sigma, \mathsf{cond}^m(Path_4), \{\mathsf{path}^m(Path_4)\}), (\bowtie, \overline{\mathsf{cond}}^m(Path_4), \{\mathsf{path}^m(Path_4)\})\}, \\ \{\mathsf{path}^m(Path_4)\}, id^m, d^m)]\end{array}} \tag{18}$$

**Output of a subtree rooted at node $\$z$** This rule is similar to the corresponding rule of Section 3.2.4 except that it handles multiple input streams and propagates the identifier $id^i$ and the DTD $d^i$ of each input stream $i$ determined in the preprocessing phase described further above.

$$\frac{}{Env \vdash \$z \Rightarrow [(\emptyset, \emptyset, \{\mathsf{path}^1(\$z)\}, id^1, d^1), \ldots, (\emptyset, \emptyset, \{\mathsf{path}^m(\$z)\}, id^m, d^m)]} \tag{19}$$

**Sequence** For each individual input stream, the sequence rule behaves just like the original rule of Section 3.2.4.

$$\frac{\begin{array}{c}Env \vdash \alpha_1 \Rightarrow [(P^1_1, A^1_1, O^1_1, id^1, d^1), \ldots, (P^m_1, A^m_1, O^m_1, id^m, d^m)] \\ \ldots \\ Env \vdash \alpha_n \Rightarrow [(P^1_n, A^1_n, O^1_n, id^1, d^1), \ldots, (P^m_n, A^m_n, O^m_n, id^m, d^m)]\end{array}}{\begin{array}{c}Env \vdash (\alpha_1, \ldots, \alpha_n) \\ \Rightarrow [(\bigcup_{i=1}^n P^1_i, \bigcup_{i=1}^n A^1_i, \bigcup_{i=1}^n O^1_i, id^1, d^1), \\ \ldots, \\ (\bigcup_{i=1}^n P^m_i, \bigcup_{i=1}^n A^m_i, \bigcup_{i=1}^n O^m_i, id^m, d^m)]\end{array}} \tag{20}$$

Similar to the rule for direct element constructors, we have again rephrased the WXQuery expression for sequences in the inference rule compared to the corresponding expression in the WXQuery definition to better support the inference rule notation.

**Example A.1** As an example for the translation of a join query into a corresponding APF, consider query $q_5$ of Figure 10(a). For both input streams of $q_5$, the translation builds a corresponding APT

just as described in Section 3.2. Additionally, the inference rules introduce a new join annotation each time they encounter a selection annotation that references elements from more than one APT. The join annotation

$$(\bowtie, \{\texttt{stream("photon1")/photons/photon/en} \\ \texttt{>= stream("photon2")/photons/photon/en + 0.5}\}, \\ \{\texttt{stream("photon1")/photons/photon/en}, \texttt{stream("photon1")/photons/photon/phc}, \\ \texttt{stream("photon2")/photons/photon/en}, \texttt{stream("photon2")/photons/photon/phc}\})$$

of $q_5$ is associated with the returned elements of all affected APTs. Figure 12(a) shows the resulting APF $f_{q_5}$ of $q_5$. □

# B Translating APFs into WXQueries

Similar to APTs, we can translate an arbitrary APF back into a corresponding WXQuery. In contrast to APTs, APFs are always structure-mutating since they represent join queries and joins are structure-mutating operators.

Figure 15 shows the query template for translating an arbitrary APF representing a join query with time-based data windows into a corresponding WXQuery. The template variables *VARi*, *STREAMi*, *PATHi*, *REFPATHi*, *SIZEi*, and *STEPi* have the same meaning as in Section 3.3. The index *i* indicates the input stream the respective template variable belongs to. The *JOINROOT* variable represents the root element name of the join result. For intermediate results created during in-network processing, we generate a generic name by concatenating the root element names of the joined input streams with underscores in between. This yields `photons_photons` in our example queries since `photons` is the root element name of both input streams. The variable *JOINPREDS* represents the join predicates. We use *PREDij* to denote the *j*-th selection predicate concerning stream *i*. The *JOINITEM* variable refers to the element name of one join result item. We generically create this name by concatenating the names of the data stream items of the joined streams. This yields `photon_photon` in our example queries since the data stream items are named `photon` in both streams. Finally, *PATHij* is the path referencing the *j*-th returned element of stream *i*, relative to *VARi*. The values of the template variables other than *JOINROOT* and *JOINITEM* are determined from an APF in a similar way as described for the translation of APTs in Section 3.3.

The `where` clause, the `if` conditions, and the *PATHi* and *PATHij* variables are optional depending on the characteristics of the corresponding APF. If any *PATHi* or *PATHij* is empty in an actual instance of the template variable, the respective preceding slash also disappears from the template. If there is no selection annotation for a certain returned element, the query simply returns the value or element without a surrounding `if` condition. In such a case, we also need to remove any `if` conditions guarding the output of the surrounding *JOINITEM* and *STREAMi* tags from the template. Each returned element is enclosed in the correct sequence of surrounding elements as in the original input stream schema, starting with the first element below the stream item, which is the `photon` element in our example stream. This is necessary to uniquely identify the elements during postprocessing and is indicated by dots in the query template of Figure 15.

**Example B.1** Figure 16 shows the abstractions of queries $q_5$ to $q_8$ of Figure 10. □

```
<JOINROOT>
  { for $VAR1 in stream("STREAM1")/PATH1|REFPATH1 diff SIZE1 step STEP1|
    ...
    for $VARm in stream("STREAMm")/PATHm|REFPATHm diff SIZEm step STEPm|
    where JOINPREDS
    return
      if (PRED11 or ... or PRED1n or ... or PREDm1 or PREDmk) then
        <JOINITEM>
          { if (PRED11 or ... or PRED1n) then
              <STREAM1>
                ...
                { if (PRED11) then $VAR1/PATH11 else () }
                ...
                { if (PRED1n) then $VAR1/PATH1n else () }
                ...
              </STREAM1>
            else () }
          ...
          { if (PREDm1 or ... or PREDmk) then
              <STREAMm>
                ...
                { if (PREDm1) then $VARm/PATHm1 else () }
                ...
                { if (PREDmk) then $VARm/PATHmk else () }
                ...
              </STREAMm>
            else () }
        </JOINITEM>
      else () }
</JOINROOT>
```

**Figure 15:** Join query template

```
<photons_photons>
  { for $x in stream("photon1")/photons/photon
      |det_time diff 10 step 5|
    for $y in stream("photon2")/photons/photon
      |det_time diff 20 step 10|
    where $x/en >= $y/en + 0.5
    return
      <photon_photon>
        <photon1>
          { $x/phc } { $x/en }
        </photon1>
        <photon2>
          { $y/phc } { $y/en }
        </photon2>
      </photon_photon> }
</photons_photons>
```

**(a)** Abstract Query 5 ($\hat{q}_5$)

```
<photons_photons>
  { for $x in stream("photon1")/photons/photon
      |det_time diff 10 step 5|
    for $y in stream("photon2")/photons/photon
      |det_time diff 20 step 10|
    where $x/en >= $y/en
    return
      <photon_photon>
        <photon1>
          { $x/en } { $x/det_time }
        </photon1>
        <photon2>
          { $y/en } { $y/det_time }
        </photon2>
      </photon_photon> }
</photons_photons>
```

**(b)** Abstract Query 6 ($\hat{q}_6$)

```
<photons_photons>
  { for $x in stream("photon1")/photons/photon
      |det_time diff 30 step 5|
    for $y in stream("photon2")/photons/photon
      |det_time diff 15 step 10|
    where $x/en >= $y/en
    return
      <photon_photon>
        <photon1>
          { $x/en } { $x/det_time }
        </photon1>
        <photon2>
          { $y/en } { $y/det_time }
        </photon2>
      </photon_photon> }
</photons_photons>
```

**(c)** Abstract Query 7 ($\hat{q}_7$)

```
<photons_photons>
  { for $x in stream("photon1")/photons/photon
      |det_time diff 15 step 10|
    for $y in stream("photon2")/photons/photon
      |det_time diff 30 step 15|
    where $x/phc >= $y/phc
    return
      <photon_photon>
        <photon1>
          { $x/en } { $x/det_time }
        </photon1>
        <photon2>
          { $y/en } { $y/det_time }
        </photon2>
      </photon_photon> }
</photons_photons>
```

**(d)** Abstract Query 8 ($\hat{q}_8$)

**Figure 16:** Abstractions of example join queries