Annex A. Design of the DQE tool

This annex presents the data model of DQE and its storage in the metabase.

1. Data model

In this section we present a simplified version of the data model of DQE, focusing in the most important classes and omitting auxiliary ones.

Framework components and sessions

The framework is composed of a set of quality graphs, a set of data sources, a set of data targets, a set of properties and a set of algorithms. The formal definitions of the framework and its components can be found in Sub-section 2.1 of Chapter 3. Sessions contains sub-sets of such components. Data sources, data targets, properties and algorithms can be contained in several sessions but quality graphs can be contained in a unique session. Figure A.1 shows framework and sessions components.

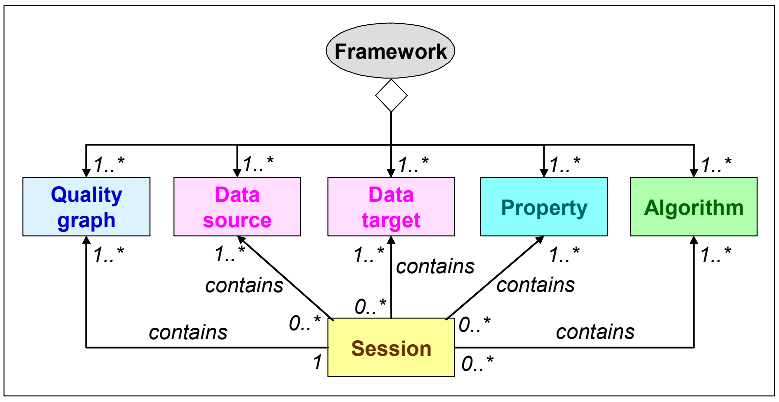

Figure A.1 – Conceptual representation of global components and session components

The remaining of the section describes the components and their inter-relations[*].

Quality graphs, data sources and data targets

Quality graphs are composed of nodes and edges. Nodes can be of three types: source nodes, target nodes, or activity nodes. The two former reference the corresponding data sources and data targets respectively. Edges relate two nodes. In order to reuse Java graph libraries, we choose to represent mono-edged graphs, i.e. graphs that have a unique edge between a pair of nodes. Edges can be of three types: data edges, control edges and mixed edges; the latter represents the existence of a data edge and a control edge between the nodes. Figure A.2 illustrates the representation of quality graphs, data sources and data targets.

[*] In the remaining figures, we color the framework components with the same colors they appear in Figure A.1 in order to quickly identify their sub-components and the relationships with other framework components.

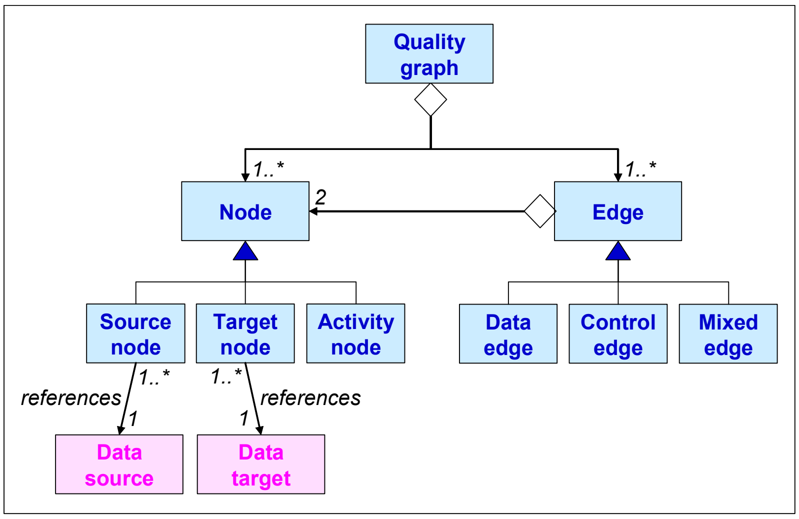

Figure A.2 – Conceptual representation of quality graphs, data sources and data targets

Properties

Properties can be of two types: features and measures. In the case of a measure, we also model the hierarchy *quality dimension* → *quality factor* → *quality metric*, as shown in Figure A.3.

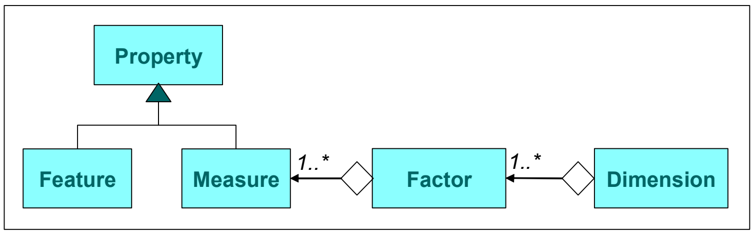

Figure A.3 – Conceptual representation of properties

The association of properties to nodes and edges is done indirectly, via groups of nodes and groups of edges. In other words, nodes and edges are grouped according to the properties they should have and properties are associated to the groups. This allows associating the relevant properties once and not for each node/edge. Figure A.4 shows the grouping of some nodes in three node groups and the association of properties to such groups; for example, as *Node*$_1$ belongs to *Group*$_1$ it has indirectly associated two properties: *Prop*$_1$ and *Prop*$_2$.

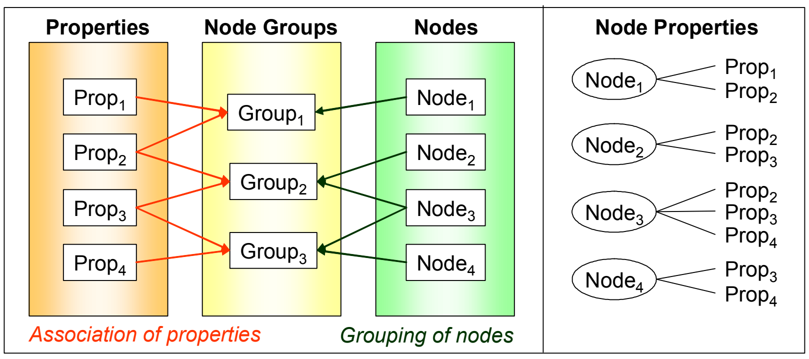

Figure A.4 – Association of properties to nodes

There are two types of associations: global and local. In the former, properties are globally associated to groups and therefore they are valid for all quality graphs whose nodes/edges belong to such groups. In the latter, the association is local to a quality graph, i.e. only the nodes/edges of the graph belonging to such groups are affected. Nodes and edges are labeled with property values (when they belong to a group that has associated the property). For generality purposes, some property values can label the whole quality graph (e.g. the label *DISadministrator='VP'*). Figure A.5a illustrates the grouping of nodes and edges and their association of property values. As data sources and data targets can also have property values, source nodes and target nodes inherit the property values of the referenced sources and targets respectively, as illustrated in Figure A.5b.

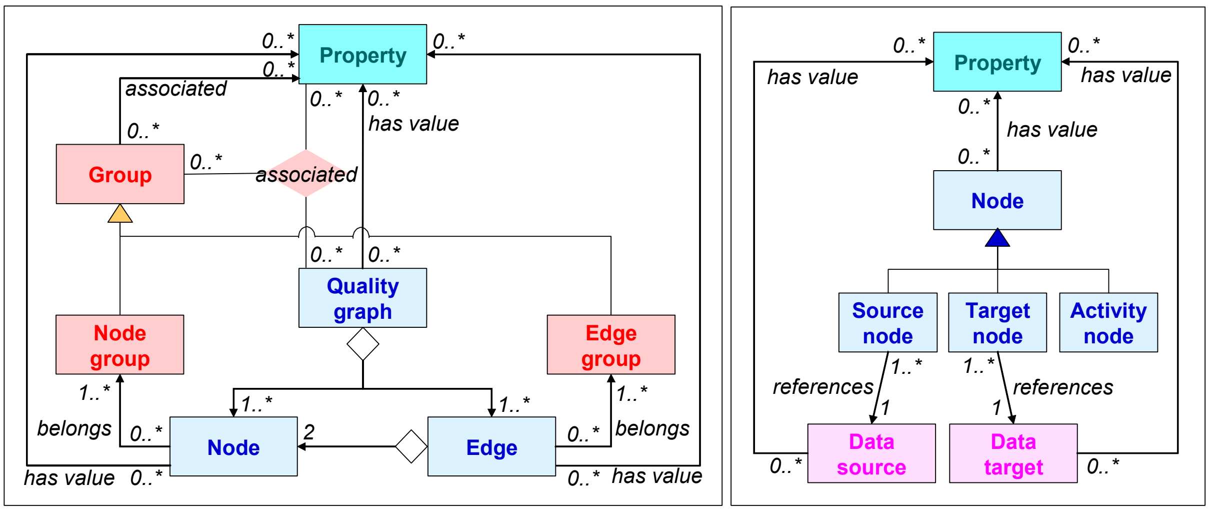

Figure A.5 – Conceptual representation of the association of property values: (a) grouping of nodes and edges, and (b) inheritance of source and target properties

Algorithms

Algorithms have associated pairs <group, property> indicating the groups that must be labelled with certain properties as precondition for executing and the groups that will be labelled with certain properties as postcondition of the execution. They also indicate the quality property that they calculate, as shown in Figure A.6.

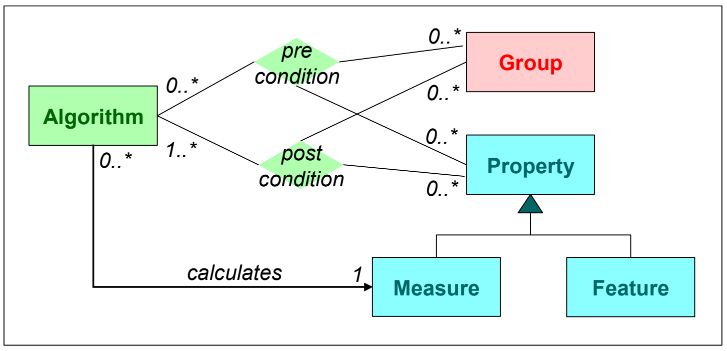

Figure A.6 – Conceptual representation of algorithms

2. Metabase

The metabase provides the persistency of the data model. The relational schema of the metabase is shown in Figure A.7; tables are colored according to the framework components that they represent, namely: 🟪 catalogues of data sources and data targets, 🟦 catalogue of quality graphs, 🟦 catalogue of properties, 🟩 catalogue of algorithms, 🟨 sessions, 🟥 grouping and 🟫 property values; bold attributes represent keys; continuous lines among relations represent foreign key dependencies while dashed arrows represent optional references (treated as foreign key depending on the value of other attributes).

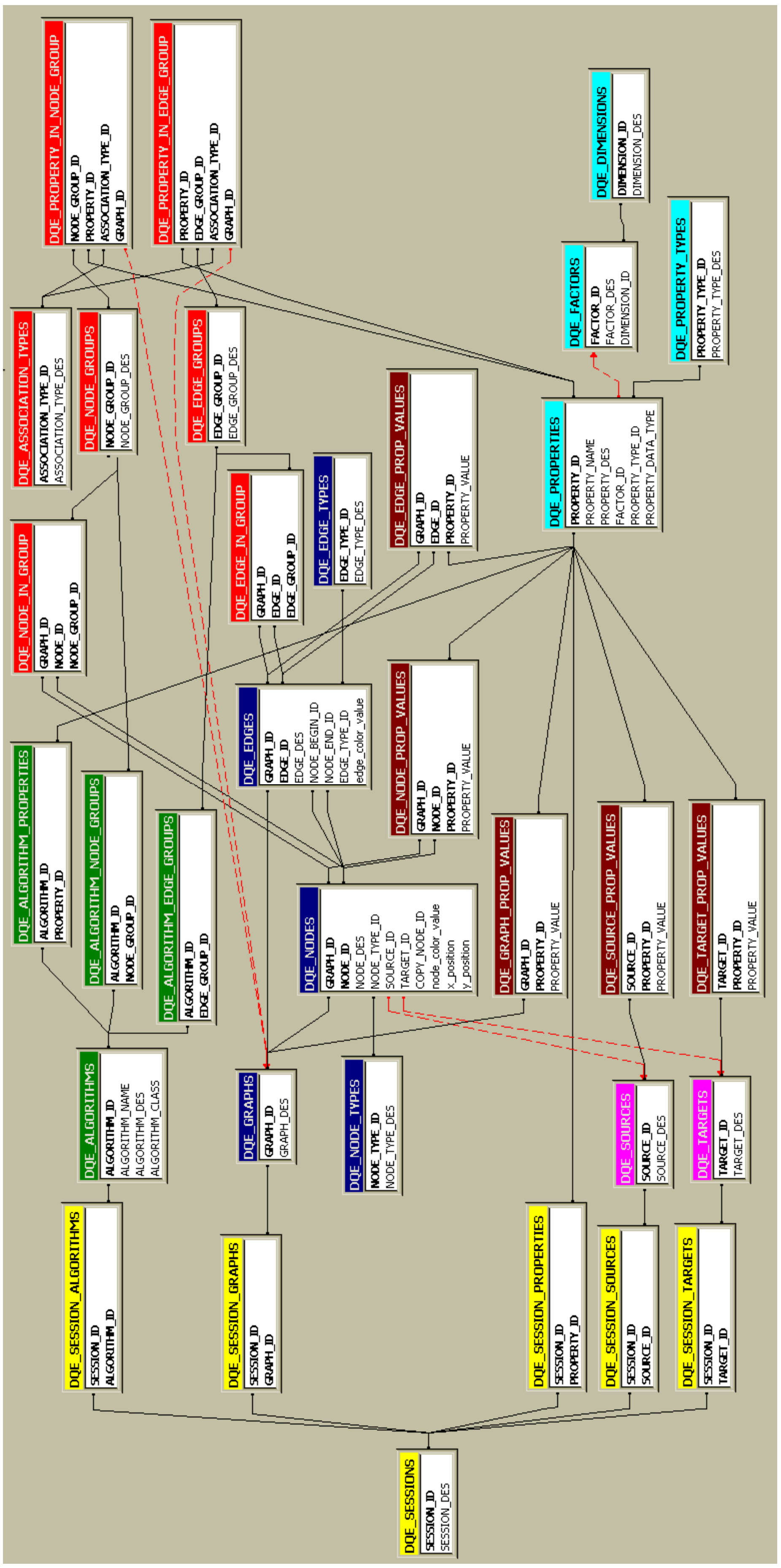

Figure A.7 – Metabase schema

Annex B. Instantiation of the Freshness Evaluation Algorithm

This annex presents the overloaded functions that instantiate the freshness evaluation algorithm in two application scenarios: a Mediation application and a Web Warehousing application.

1. Mediation application scenario

In this section we recall the relevant properties of the mediation application scenario and we present the pseudocodes of the corresponding overloaded functions.

Four freshness evaluation algorithms where proposed:

- *TimelinessEvaluation1*: It is the algorithm used in virtual contexts or when user expectations range several months. Processing costs and inter-process delays (including those caused by data materialization) are neglected. Source data actual timeliness is considered.
- *TimelinessEvaluation2*: It is the algorithm used in materialization contexts when user expectations range several days or weeks. Inter-process delays due to data materialization are considered, as well as source data actual timeliness. Processing costs and other inter-process delays are neglected.
- *CurrencyEvaluation1*: It is the algorithm used in virtual contexts. Processing costs are estimated from the number of input tuples and inter-process delays are neglected. Source data actual currency is neglected.
- CurrencyEvaluation2: It is the algorithm used in materialized contexts. Processing costs and inter-process delays among operation nodes are irrelevant compared to refreshment periods, hence, the unique property value that is consider is the inter-process delay cause by data materialization.

Table B.1 recalls the calculation of properties, which was discussed in Sub-section 3.1.3 of Chapter 5 (Table B.1 is adapted from Table 5.1).

	Data timeliness 1	**Data timeliness 2**	**Data currency 1**	**Data currency 2**
Processing cost (A) one predecessor B	Neglect	Neglect	*Tuples*(B) / *Capacity*(A)	Neglect
Processing cost (A) two predecessors B_1 and B_2	Neglect	Neglect	*Tuples*(B_1) * *Tuples*(B_2) / *Capacity*(A)	Neglect
Inter-process delay (between operators)	Neglect	Neglect	Neglect	Neglect
Inter-process delay (between the last operator A and the interface I)	Neglect	*Refreshment period* (A)	*Refreshment period* (A)	*Refreshment period* (A)
Source data actual freshness (S), successor A	*Update period* (S)	*Update period* (S)	Neglect	Neglect
Combination of input values	Maximum of input values	Maximum of input values	Maximum of input values	Maximum of input values

Table B.1 – Calculation of property values with different types of estimation

Algorithm B.1 shows the pseudocodes of the overloaded functions for the second algorithm (timeliness 2 in Table B.1). The *getSourceActualFreshness* function returns the source update period. The *getProcessingCost* function returns zero because processing cost is neglected. The *getInterProcessDelay* function returns zero for all edges except that incoming the mediator interface, which is calculated as the refreshment period. The *combineActualFreshness* function returns the maximum of input values.

For the first algorithm (timeliness 1), the *getInterProcessDelay* function returns zero; the other functions are reused from Algorithm B.1.

For the fourth algorithm (currency 2), the *getSourceActualFreshness* function returns zero; the other functions are reused form Algorithm B.1.

For the third algorithm (currency 1), the *getProcessingCost* function is calculated from the tuples and capacity properties, as shown in Algorithm B.2; the other functions are reused form Algorithm B.1.

```
FUNCTION getSourceActualFreshness (G: QualityGraph, A: Node) RETURNS INTEGER
    INTEGER value = G.getPropertyValue(A,"UpdatePeriod");
    RETURN value;
END

FUNCTION getProcessingCost (G: QualityGraph, A: Node) RETURNS INTEGER
    RETURN 0;
END

FUNCTION getInterProcessDelay (G: QualityGraph, e: Edge) RETURNS INTEGER
    NODE A= e.source, B= e.target;
    IF G.belongsToGroup(B,"Interface") THEN
        INTEGER value = G.getPropertyValue(A,"RefreshPeriod");
        RETURN value;
    ELSE
        RETURN 0;
END

FUNCTION combineActualFreshness (G: QualityGraph, valList: HushTable) RETURNS INTEGER
    INTEGER aux= 0;
    FOR EACH <e, value> in valList DO
        IF value > aux
            aux = value;
    ENDFOR
    RETURN aux;
END
```

Algorithm B.1 – Overloading of functions for the second algorithm (timeliness 2)

```
FUNCTION getProcessingCost (G: QualityGraph, A: Node) RETURNS INTEGER
    INTEGER tuples;
    INTEGER capacity = G.getPropertyValue(A,"Capacity");
    LIST OF NODE nList = G.getPredecessors (A);
    IF (nList.getSize == 1) THEN
        NODE B= nList.getFirst();
        tuples = G.getPropertyValue(B,"Tuples");
        RETURN tuples / capacity;
    ELSE
        NODE B1= nList.getFirst();
        tuples = G.getPropertyValue(B1,"Tuples");
        NODE B2= nList.getLast();
        tuples = tuples * G.getPropertyValue(B2,"Tuples");
        RETURN tuples / capacity;
END
```

Algorithm B.2 – Overloading of the getProcessingCost function for the third algorithm (currency 1)

2. Web warehousing application scenario

In this section we recall the relevant properties of the web warehousing application scenario and we present the pseudocodes of the corresponding overloaded functions.

Table B.2 recalls the calculation of properties, which was discussed in Sub-section 3.2.3 of Chapter 5 (Table B.2 is adapted from Table 5.2 and Table 5.3).

	Precise value	**Average case**	**Worst case**
Processing cost (A)	Neglect	Neglect	Neglect
Inter-process delay (A,B), B in {i_1, i_2, t_1, t_2, t_4, t_5}	Neglect	Neglect	Neglect
Inter-process delay (A,B), B in {m_1, m_2, t_3, v_1, v_2, v_3, v_4, v_5}	*Last execution time* (B) – *Last execution time* (A)	Average in statistics of: *Execution time* (B) – *Execution time* (A)	Maximum in statistics of: *Execution time* (B) – *Execution time* (A)
Source data actual currency (S)	Neglect	Neglect	Neglect
Source data actual timeliness (S), push	Neglect	Neglect	Neglect
Source data actual timeliness (S), periodic pull, wrapper W	*Last execution time* (W) – *Last change detection* + *Pull period* (W)	*Pull period* (W) + Average in statistics of: *Execution time* (W) – *Change detection time*	*Pull period* (W) + Maximum in statistics of: *Execution time* (W) – *Change detection time*
Combination of input values (different data volatility)	Input value of most volatile input		
Combination of input values (equal data volatility)	Average of input values weighted with *Data volume*	Average in statistics of: average of input values weighted with *Data volume*	Maximum (input values)

Table B.2 – Calculation of property values with different types of estimation

Algorithm B.3 shows the pseudocodes of the overloaded functions for calculating timeliness with the precise estimation strategy, according to Table B.2. The *getSourceActualFreshness* function returns different values depending on the source capabilities, namely it returns: (i) zero for sources that can announce changes and (ii) the difference between wrapper execution time and last change detection time plus the pull period of the wrapper for the other sources. The *getProcessingCost* function returns zero because processing cost is neglected. The *getInterProcessDelay* function returns different values for the different types of edges, namely it returns: (i) zero when the successor activity belongs to the *NoDelay* group (i.e. activities i_1, i_2, t_1, t_2, t_4 and t_5) and (ii) the difference of execution times for the other activities. The *combineActualFreshness* function traverses the list of input values keeping the total (sum of input values multiplied by data volumes) and volume (sum of data volumes) variables, which allows the calculation of the weighted average at the end. When a more volatile predecessor is found, such values are reinitialized. Note that some property values are obtained from the graph (labels of nodes and edges) but other ones are read in a log.

For calculating currency, the *getSourceActualFreshness* function returns zero, and the other functions are reused from Algorithm B.3.

The pseudocodes of the overloaded functions for the average and worst case estimation strategies are analogous. The difference is that the log methods *getAverage* or *getMaximum* are invoked instead of the *getLast* method (highlighted in red in Algorithm B.3).

```
FUNCTION getSourceActualFreshness (G: QualityGraph, A: Node) RETURNS INTEGER
    INTEGER announce= G.getPropertyValue(A,"AnnounceChanges")
    IF announce = TRUE THEN
        RETURN 0;
    ELSE
        NODE W= G.getSuccessors(A)  /*the wrapper*/
        LOG log = G.getLog("Execution-ChangeDetection");
        INTEGER time = log.getLast(A,W);
        INTEGER pull= G.getPropertyValue(W,"PullPeriod");
        RETURN time + pull;
END
```

```
FUNCTION getProcessingCost (G: QualityGraph, A: Node) RETURNS INTEGER
    RETURN 0;
END
```

```
FUNCTION getInterProcessDelay (G: QualityGraph, e: Edge) RETURNS INTEGER
    NODE A= e.source, B= e.target;
    INTEGER time;
    IF G.isSource(A) or G.isTarget(B) THEN
        RETURN 0;
    ELSE IF G.belongsToGroup(B,"NoDelay") THEN
        RETURN 0;
    ELSE
        LOG log = G.getLog("Execution-Execution");
        INTEGER time = log.getLast(A,B);
        RETURN time;
END
```

```
FUNCTION combineActualFreshness (G: QualityGraph, valList: HushTable) RETURNS INTEGER
    INTEGER maxVolatility= 0, volume= 0, total= 0, volatility, aux;
    FOR EACH <e, value> in valList DO
        NODE A= e.source;
        volatility = G.getPropertyValue(e,"DataVolatility");
        LOG log = G.getLog("DataVolumne");
        INTEGER aux = log.getLast(A);
        IF volatility > maxVolatility THEN
            maxVolatility = volatility;
            volume = aux;
            total = aux * value;
        ELSE IF volatility = maxVolatility THEN
            volume = volume + aux;
            total = total + aux * value;
    ENDFOR
    IF volume = 0 THEN RETURN 0;
    ELSE RETURN total / volume;
END
```

Algorithm B.3 – Overloading of functions for timeliness and precise estimation strategy