# Chapter 5. Experimentation and Applications

*This chapter presents our experimentation results.*
*We describe several applications and the use of the quality evaluation framework for evaluating data quality in such applications.*
*We describe a prototype of a Data Quality Evaluation tool, DQE, which implements the framework and allows the execution of quality evaluation algorithms.*
*The prototype was used for evaluating data freshness and data accuracy in several application scenarios.*

## 1. Introduction

Chapters 3 and 4 described our proposal for the evaluation of data freshness and data accuracy in data integration systems. In this chapter we illustrate its practical use in real applications. To this end, we have developed a prototype of an evaluation tool, called DQE, which implements the proposed quality evaluation framework. The tool allows displaying and editing the framework components as well as executing quality evaluation algorithms.

The prototype was used for evaluating data freshness and data accuracy in several application scenarios in order to validate the approach. Specifically, we describe three applications: (i) an adaptive system for aiding in the generation of mediation queries, (ii) a web warehousing application retrieving movie information, and (iii) a data warehousing system managing information about students of a university. We briefly describe each application, we model it in DQE (quality graphs, properties, etc.) and we explain the evaluation of data freshness and data accuracy for them.

The goal of our experimentation is twofold. Firstly, we want to validate the approach in real applications, analyzing the practical difficulties of modeling different DISs as quality graphs, setting property values and instantiating evaluation algorithms for them. Secondly, we want to test the execution of evaluation algorithms for such quality graphs. For the first application (mediation application) we also tested the connection of the tool with other DIS design modules, communicating via a metadata repository.

In addition, we describe some tests for evaluating performance and limitations of the tool. To this end, we generated some data sets (quality graphs adorned with property values) and we executed a quality evaluation algorithm over each graph. The test results allow affirming that the tool can be used for large applications (modeling hundreds of graphs with hundreds of nodes each).

The following sections describe our experimentations: Section 2 presents the DQE tool, describing its functionalities, architecture and interface. Section 3 uses DQE in three application scenarios, describing the evaluation experiments performed in each application and their results. Section 4 presents the performance tests, describing the generation of test data sets, the tests themselves and the obtained results. Finally, section 5 concludes.

## 2. Prototype

Most of the functionalities of the framework for data quality evaluation described in previous chapters have been implemented in a quality auditing tool called *DQE* (Data Quality Evaluation). The prototype was implemented in Java (JDK 1.4) and manages persistence via an Oracle® database (Oracle 10g Enterprise Edition Release 10.1.0.2.0). The tool allows displaying and editing the framework components as well as executing quality evaluation algorithms. Next sub-sections describe the tool main characteristics; design and implementation features are treated in Annex 1.

## 2.1. Functionalities

The main functionalities of the tool are:

- *Storage of framework components*: The framework components (data sources, data targets, quality graphs, properties and algorithms), auxiliary components that ease the manipulation of framework (e.g. sessions, groups of nodes and edges) and graphical components, are all stored in a relational database, called *metabase*. The metabase contains global catalogues for the various components and maintains association lists among components. The goal of the metabase is threefold: it provides persistency, it allows reusing components previously defined, and it serves as communication media for sharing data with other DIS design tools (in particular with the tool managing user profiles, which stores user expected quality values) and statistical tools (which estimate and store property values).
- *Management of sessions*: A session represents a concrete application scenario and encloses the framework components for that scenario, i.e. data sources, data targets, quality graphs, properties and algorithms. Several sessions can be stored in the metabase but the tool only manages one session at a time. Among the functionalities of management of sessions, the tool includes: the creation, loading, storing and deletion of sessions, the creation, loading, storing and deletion of components. Most of the components (data sources, data targets, properties and algorithms) are stored in global catalogues of the metabase and can be loaded in several sessions. Quality graphs, however, are associated to a unique session.
- *Management of properties*: An editor of properties allows the creation, modification and deletion of properties from the global catalogue and their inclusion/removal from the current session. Information describing properties include: a code (auto-generated), a name (globally unique, used by the user for identifying the property), a description (natural language description of the property semantics and units), a type (feature or measure) and the domain data type (e.g. *integer*). In the case of measure properties, the dimension and factor and also required (e.g. the *Currency* metric corresponds to the *Currency* factor of the *Freshness* dimension).
- *Management of sources and targets*: Sources and targets are managed in the same way. A simple editor allows the creation, modification and deletion of sources/targets from the global catalogue and their inclusion/removal from the current session. Information describing sources/targets include: a code (auto-generated) and a description. Sources may have associated some property values (e.g. actual quality values, availability, access cost); targets may also have associated some property values (e.g. expected quality values, workload). The editor also allows associating and dissociating properties to sources and targets as well as changing property values.
- *Management of algorithms*: A simple interface allows the inclusion and deletion of algorithms from the global catalogue and their inclusion/removal from the current session. An algorithm is identified by a name (globally unique) and has a description and a reference to a Java class that implements it. The current version does not provide a graphical interface for implementing an algorithm; algorithm code should be produced using a text editor or a Java development environment. However, the tool provides a method for dynamically adding new algorithms (automatically binding arguments and properties) without closing the session.
- *Visualization and edition of quality graphs*: Quality graphs are the most important components managed by the application and consequently, a sophisticated display utility, called *graph viewer*, allows visualizing the graphs and their components (nodes, edges, labels). The graph viewer allows visualizing nodes and edges, moving them in order to better analyze the graph topology, showing or hiding property values, grouping nodes and edges, coloring nodes and edges according to different criteria and zooming. The graph viewer also allows adding and removing nodes and edges to a quality graph, and changing property values.
- *Association of properties to nodes and edges of a quality graph*: Nodes and edges are grouped in order to associate properties to them (for example: we can define the group *MaterializedActivities* and associate the property *RefreshFrequency* to all nodes in such group). The tool provides functionalities for creating and deleting groups of nodes and edges, inserting and deleting nodes and edges to a group and associating and dissociating properties from a group. Source and target nodes also inherit the properties of the corresponding source or target. As a node or edge can belong to several groups, they can have associated a same property several times, but a unique value is set.

- *Execution of an algorithm*: There are two ways of executing an algorithm. The first one consists in selecting a quality graph and invoking an algorithm (selected from the session algorithms) for the graph. The second one consists in invoking an algorithm for all the quality graphs of the session. The execution of an algorithm in a quality graph is performed in a separate thread, so other functionalities can be invoked during the execution of an algorithm. The execution of an algorithm on different graphs is parallelized but the execution of several algorithms on a same graph is sequential.
- *Visualization of results*: The graph viewer is automatically refreshed after the execution of an algorithm, showing the new property values (corresponding to the evaluated quality factor). The auditing tool allows coloring nodes that do not achieve quality expectations, identifying and coloring critical paths, changing property values in order to test alternative configurations and re-executing the evaluation algorithms to see the effects of the changes.

## 2.2. Architecture

The architecture is structured in layers and the communication among layers is done via interfaces in order to standardize the access to each layer. The design of the graphical interface follows the *model-view-controller* pattern, implemented in three different layers (*View*, *Logic* and *Model*). The persistency in a relational database (Oracle 10g) is implemented in the *DataAccess* layer. Exception management and other utility functions are accessed via the *Utilities* transversal layer. Figure 5.1 shows the layers of the architecture and the interaction among them, as well as the most relevant packages that compose each layer.

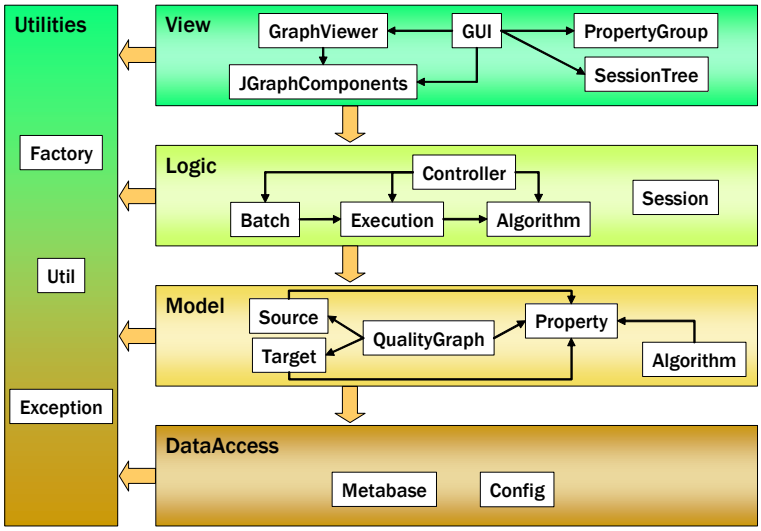


**Figure 5.1 – Architecture of the prototype**

The *View layer* manages the graphical representation. The main packages are: *GUI* (controls the graphical user interface), *GraphViewer* (displays and allows the creation and edition of quality graphs), *JGraphComponents* (manages graph components (nodes, edges, labels) and their graphical properties), *SessionTree* (manages session components) and *PropertyGroup* (manages the properties and groups associated to nodes and edges).

*Logic layer*: Implements the application logic, bringing methods for managing sessions and executing algorithms. The main packages are: *Controller* (acts as a bridge between graphical interface and data model), *Session* (manages sessions), *Algorithm* (manages the collection of algorithms, their arguments, preconditions and implemention), *Execution* (controls the execution of algorithms) and *Batch* (controls batch execution of algorithms over all quality graphs).

*Model layer*: Manages the data model. Tha main packages contain the representation of the framework components: *Source*, *Target*, *QualityGraph*, *Property* and *Algorithm*.

*DataAccess layer*: Persists the data model in a relational database (*Metabase* package) and interacts with configuration files (*Config* package).

*Utilities layer*: Brings basic services to the other layers. The main packages are: *Exception* (handles exceptions), *Factory* (manages the creation of objects) and *Util* (contains utilitarian functions).

## 2.3. Interface

The graphical interface of the tool is shown in Figure 5.2. The main window consists of four main components:

- *Main menu* (top): The menu allows adding, deleting and editing session components, global components and groups.
- *Graph Viewers panel* (right): A *Graph Viewer* is the editor of a quality graph. It allows displaying a graph, adding/deleting nodes and edges, inserting/deleting groups, changing property values, showing/hiding properties and groups, coloring nodes and edges according to different criteria, zooming and highlighting critical paths. The *Graph Viewers panel* is a container for displaying *Graph Viewers*; it can show several graphs allowing its comparison. In Figure 5.2, the *Graph Viewer* of *Graph1* is active and the *Graph Viewers* of *Graph2* and *Graph3* are iconified.
- *Session Tree panel* (up-left): Shows the components of the current session, structured as a tree, and controls the insertion/deletion of components. The visualization of *Graph Viewers* and the execution of evaluation algorithms are also invoked from this panel. As shown in Figure 5.2, *Session1* is composed of three quality graphs, five data sources, five data targets, two evaluation algorithms and five properties.
- *Group-Property panel* (down-left): When a set of nodes/edges of a quality graph is selected in the *GraphViewer*, their groups and properties are shown in the *Group-Property panel*. The panel also allows inserting/deleting the selected nodes or edges to groups, changing property values, and indicating the properties that are shown/hidden in the *GraphViewer* (for readability purposes). In Figure 5.2, the edge A1-A5 is selected in the *GraphViewer* of Graph1 and consequently its groups and properties are listed in the *Group-Property panel*.

A user manual (in Spanish) can be found in [Ramos+2006].

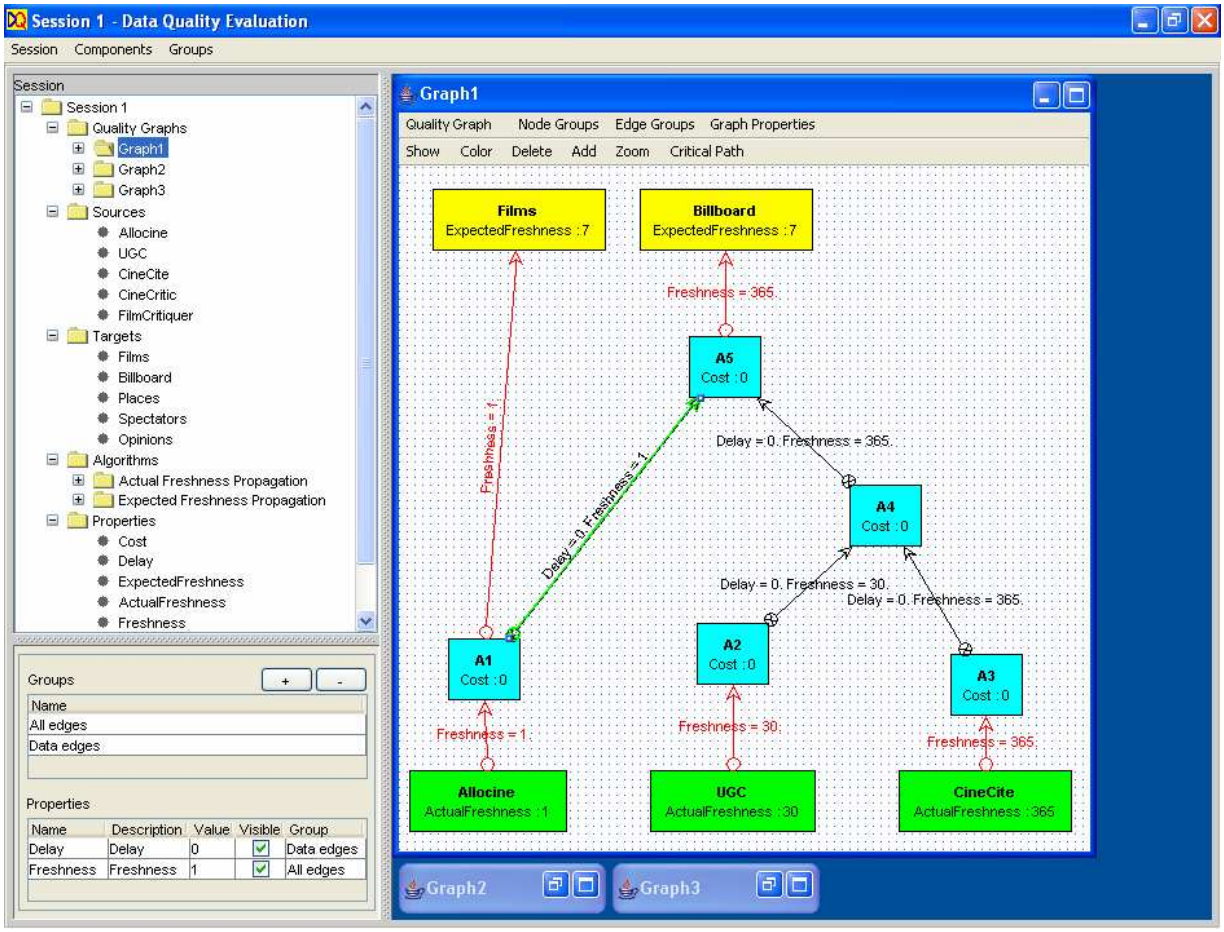


**Figure 5.2 – Graphical Interface**

## 2.4. Practical use of the tool

In this section we describe the typical use of the DQE tool for data quality evaluation, not excluding that other sequences of actions can be followed in particular application scenarios and that individual functionalities can be used for other purposes (e.g. visualizing graphs representing other aspects of an application). Therefore, the quality evaluation process using DQE can be abstracted as consisting in two major phases: (i) the personalization of the evaluation algorithms and (ii) their use for evaluating data quality in a DIS. Figure 5.3 illustrates the steps in the quality evaluation process.

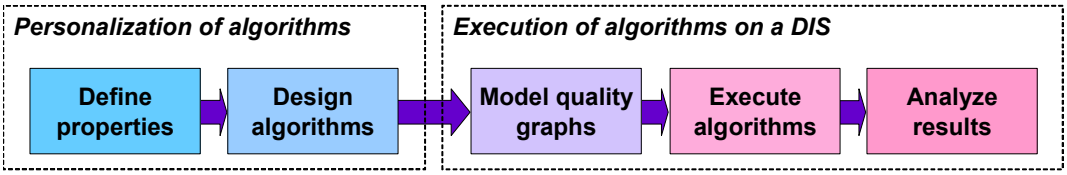


**Figure 5.3 – Steps in the quality evaluation process**

In the first phase, the user must define (or reuse from the catalogue of properties) the desired quality factors and define (or reuse) the properties that have impact in such quality factors. The tool has a default catalogue of properties that includes all properties mentioned in Chapter 3 and Chapter 4, but the user can add its own properties.

Having defined properties, evaluation algorithms must be implemented, for propagating quality combining such property values. The algorithms for propagating data freshness and data accuracy described in previous chapters, as well as other test algorithms, are included in the default catalogue. In addition, the tool provides an interface and a set of primitive methods for easing the implementation of new algorithms. The user can obtain an evaluation algorithm in three ways:

- Selecting a default propagation algorithm provided by DQE. The property values needed by the algorithm (e.g. source data actual freshness, processing costs and inter-process delays required for the *ActualFreshnessPropagation* algorithm) must be calculated using external methods and stored in the metabase (or manually set in the GraphViewer interface).
- Extending an existing propagation algorithm by overloading the methods for calculating property values. In this case, a new algorithm must be implemented as a Java class extending a default propagation algorithm and overloading the appropriate functions (e.g. overloading the getSourceActualFreshness, getProcessingCost and getInterProcessDelay methods of the *ActualFreshnessPropagation* algorithm, as described in Sub-section 3.5 of Chapter 3).
- Implementing a new algorithm from scratch. To this end, we defined a Java interface (*IAlgorithm*) whose unique method (*execute*) is invoked by DQE for executing the algorithm. The new algorithm must be implemented in a new Java class that implements the *IAlgorithm* interface and the propagation strategy must be implemented in the *execute* method. All the methods of the *QualityGraph* class can be invoked in order to traverse a quality graph with the desired propagation strategy.

In the second phase, the algorithms are used for evaluating the quality of a set of a set of quality graphs. Firstly, a session must be created, selecting the previously defined properties and algorithms, and creating (or reusing) the data sources, data targets and quality graphs. The GraphViewer can be used for creating quality graphs from scratch, editing existing quality graphs or cloning existing quality graphs (e.g. for testing different property values).

Once quality grahs are modeled and property values are associated to their nodes and edges, the evaluation algorithms can be executed. DQE allows executing an algorithm on a selected qualtiy graph or executing an algorithm (in parallel) on all the quality graphs of the session.

After executing algorithms, the GraphViewer can be used again for analyzing evaluation results, for example, highlighting critical paths. Several quality graphs (for example representing alternative implementations of the DIS) can be graphically compared. The evaluation results are also stored in the metabase in order to support decision-making using external tools. In order to support what-if analysis, some property values can be changed (simulating alternative configurations) and the algorithms can be re-executed with the new property values.

### 2.5. Liberation of versions

The tool first liberation dates from April 2004. The second version, liberated in October 2004, was used for the tests and results that will be discussed in next sections. However, the implementation of the tool was continued, improving persistency and graphical interface, resulting in two additional liberations. Further improvements are scheduled, which are discussed as perspectives in Chapter 6.

History of versions and implementers:

- Version 1 – April 2004: Initial design of the framework; emphasis in the design of the overall architecture and its extensibility (dynamic incorporation of quality properties and evaluation algorithms). Implementation of the main framework components with rapid solutions for evaluation algorithms (basic test algorithms), persistency (via XML files) and graphical interface (ad-hoc components). Developed in a pre-grade project by Fabian Fajardo and Ignacio Crispino, supervised by Verónika Peralta and Raúl Ruggia [Fajardo+2004], presented in the CACIC'2004 conference [Fajardo+2004a].
- Version 2 – October 2004: Incorporation of new functionalities, emphasis in the interoperation with other DIS design tools via a metadata repository. Implementation of freshness evaluation algorithms, incorporation of new display functionalities (as visualization of critical paths), enrichment of the graphical interface and persistence of the framework in the metadata repository. Developed by Verónika Peralta, presented in the BDA'2004 conference [Kostadinov+2004].
- Version 3 – August 2005: Reengineering of data and persistency modules, emphasis in scalability, replacing memory storage by database accesses. Extension of the framework data model, including new features (as groups of nodes and edges) and graphical properties (as colors of nodes and edges), and implementation of the data model in a relational database. Rapid solutions for graphical interface. Developed in a pre-grade project by María José Rouiller, supervised by Verónika Peralta [Rouiller+2005].
- Version 4 – April 2006: Reengineering of the graphical interface, emphasis in easing user interaction. Restructuration of menus and toolbars, incorporation of new display functionalities and implementation of some accuracy evaluation algorithms. Developed in a pre-grade project by Mayra Ramos and Renzo Settimo, supervised by Verónika Peralta, Adriana Marotta and Salvador Tercia [Ramos+2006].

Section 3 describes several application scenarios and the use of the tool for evaluating data freshness and/or data accuracy in them.

## 3. Applications

In this section we describe some application scenarios where we used the quality evaluation framework. We briefly describe each application scenario, we model it in DQE and we illustrate quality evaluation for it.

### 3.1. An adaptive system for aiding in the generation of mediation queries

Nowadays, mediation systems are well-known and there exists a great number of implementations. Their main components are: the global schema, mappings between the global schema and the data sources, query rewriting functions and result merging functions. All these components take into account the heterogeneity of data sources which is one of the main problems treated by mediation systems. Other design problems appear at mediator exploitation time. Among these problems we distinguish the definition of the mappings between the global schema and the data sources. Because of a great number of data sources, possibly containing redundant information and different data quality, it is also important to adapt these mappings to user's needs, in particular, in terms of data quality.

In [Kostadinov+2005], we presented an adaptive system for aiding in the generation of *mediation queries* (which represent the mappings between the mediation schema and the data sources). The goals of the system are, on the one hand, to automatically generate mediation queries taking into account data heterogeneity, and on the other hand, to adapt the queries to the user requirements in terms of quality. In this sub-section we describe the system and the use of DQE in the design of mediation applications adapted to user preferences.

The goal of this experiment is twofold: Firstly, we want to validate our approach, instantiating the framework and evaluating data freshness in this concrete application scenario. Secondly, we want to test different evaluation algorithms adapted to different configurations of the system (e.g. materializing data or not).

### 3.1.1. Description of the application

A mediation system is defined as the integration of several distributed and heterogeneous data sources. The integration is described by means of a global schema, called *mediation schema*, and the mappings relating it to the data sources, called *mediation queries*. Figure 5.4 presents an overview of the architecture of mediation systems.

The generation of mediation queries is one of the most tedious task to be done manually, because of the great number of sources that may be involved (hundreds or thousands) and of the volume of the metadata describing them (source and global schema descriptions, semantic correspondence assertions, etc.). As there may exist several sources providing the same type of data, several mediation queries can define a same mediation object. A key problem is determining the sources (or combination of sources) that provide the results the most adequate to user's needs, according to their thematic and quality preferences. In this context, data quality has a fundamental role in the design and exploitation of mediation applications.

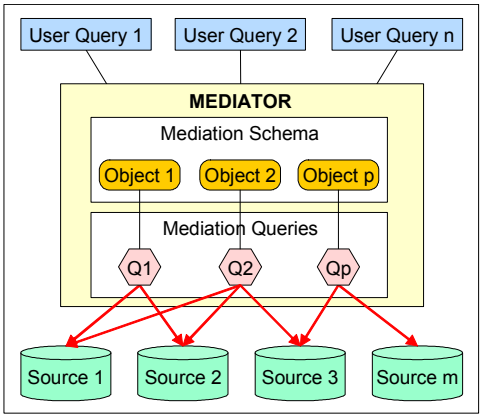


**Figure 5.4 – Mediation system architecture**

**Example 5.1.** Consider a mediation system providing information about scientific publications and their authors. The mediation schema is composed of a unique relation:

Publications (**title**, **author**, affiliation, conference, year, editor)*

Also consider four data sources: AuthorBase, ConfList, DBpubs and LP. AuthorBase contains information about authors of scientific publications, ConfList is a catalogue of conferences, DBpubs and LP are lists of published articles. Their schemas are:

- Source AuthorBase : Authors (**author**, address, email, affiliation, nationality)
- Source ConfList : Conferences (**conference**, year, city, country, editor)
- Source DBpubs : Publications (**author**, **title**, conference)
- Source LP : Pubs (**author, title**, conference, year, editor)

In order to calculate the mediation relation the source relations must be combined. In this example, there is not a unique solution. Queries $Q_1$, $Q_2$, $Q_3$ (and their combinations: $Q_1 \cup Q_2$, $Q_1 \cap Q_3$, etc.), allows to obtain the mediation relation:

- $Q_1$ = LP. Pubs $\bowtie_{author}$ AuthorBase.Authors
- $Q_2$ = DBpubs.Publications $\bowtie_{author}$ AutorBase.Authors $\bowtie_{conference}$ ConfList.Conferences
- $Q_3$ = DBpubs.Publications $\bowtie_{author}$ AutorBase.Authors $\bowtie_{auteur,\ titre}$ LP.Pubs

Even if each query allows providing all the attributes of the mediation relation, they produce results with different semantics and different quality. For example, if the source DBpubs is not frequently updated, it can provide data that is less fresh than LP, being less interesting for a researcher searching for new publications. □

* Attributes in bold constitute the key of the relations

The application consists in generating several mediation queries, evaluating the quality of their data and selecting the most appropriate one according to user preferences. A design toolkit provides these functionalities. It is composed of three tools: *mediation query generation*, *user profile management* and *data quality evaluation*. The tools communicate via a metabase server which store all the metadata used and produced by the tools (metadata describing sources, the mediator, mappings, quality measures and user profiles). A definition interface allows users to interact with the tools. Figure 5.5 shows the architecture of the toolkit.

The *mediation query generation* tool [Xue 2006] is responsible for the selection of relevant sources, the detection and resolution of semantic conflicts and the generation of mediation queries. The *user profile management* tool [Kostadinov 2006] is responsible for the definition of user profiles, which include quality expected values. The *data quality evaluation* tool (presented in Section 2) is responsible for the estimation of the quality of data provided by the generated queries and the selection of those satisfying user quality expectations.

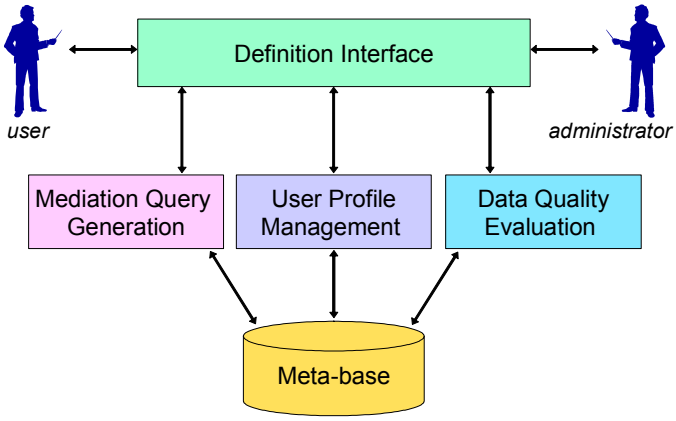


**Figure 5.5. Toolkit architecture**

In the remaining of this sub-section we discuss quality evaluation features; an overview of the functionalities of the other tools can be found in [Kostadinov+2005].

### 3.1.2. Modeling of the application in DQE

In this sub-section we illustrate the instantiation of the quality evaluation framework for the application introduced in previous sub-section. Concretely, we model each generated mediation query as a quality graph, allowing the execution of quality evaluation algorithms on the graphs and the comparison among them.

Each query operator is represented as an activity. An additional activity represents the mediator interface for user queries. Figure 5.6 shows a quality graph for the mediation query $Q_3$ of Example 5.1.

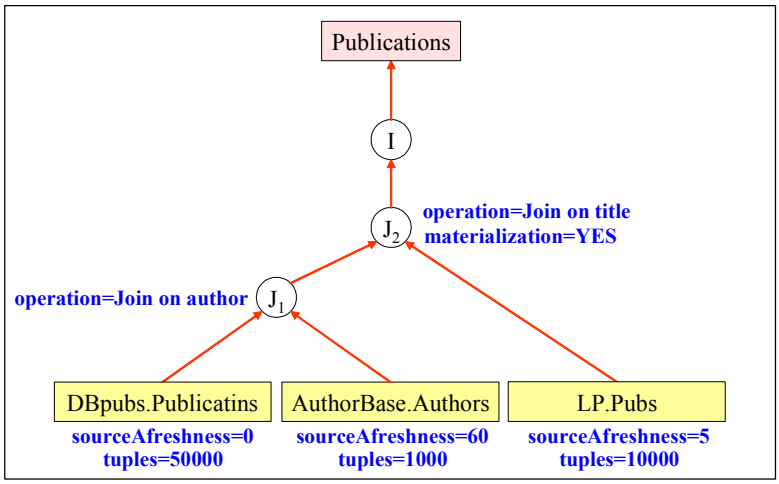


**Figure 5.6 – Quality graph for a mediation query**

We build two quality graphs for each mediation query, having the same topology but differing in some property values. Concretely, one of the graphs represents a virtual mediation query while the other represents the materialization of the query data. Consequently, synchronization techniques include synchronous and asynchronous pull policies.

Considering that users are interested in measuring both, currency and timeliness, the application scenario is characterized as follows:

- **MED scenario:**
  - Freshness factor: currency and timeliness
  - Nature of data: depends on specific sources (all types of data are possible)
  - Architectural techniques: virtual and materialization techniques
  - Synchronization policies: pull-pull and pull/pull policies

The estimation of property values, taking into account the characteristics of the scenario, is discussed in next sub-section.

### 3.1.3. Estimation of property values

In order to manipulate concrete freshness expectations, we take as case of study, the domain of scientific publications and their authors introduced in Example 5.1 and we assume that users have very different freshness expectations, ranging from "some days" to "several months" for timeliness and ranging in "some minutes" for currency. Therefore, we need several evaluation algorithms, adapted to different DIS properties and different user expectations.

For timeliness requirements, the "day" is a good unit for measuring data timeliness and properties values ranging in "some hours" or less can be neglected. Query operation costs and inter-process delays among operations are negligible compared with timeliness expectations. However, when materializing data, the inter-process delays with mediator interfaces are relevant. Such delays can be estimated, in the worst case, as the refreshment period. Source data actual timeliness is also relevant. Data timeliness is measured, in the worst case, as the source update period. We assume that external processes are capable of estimating and bounding the time passed since last source update, and that such processes store the calculated value as a source property.

For currency requirements, the "second" should be used for measuring data currency and properties values ranging in "some seconds" should be considered. Query operation costs are relevant. As our purpose was to test the execution of evaluation algorithms and not to compare sophisticated cost models, we simply estimated processing costs based on the number of tuples of source relations. Concretely, the processing cost was computed dividing the number of input tuples (product of numbers of tuples of both input relations, in the case of a join) by the number of tuples that can be processed per second (operation capacity). Note that we need to estimate the number of tuples resulting from each operator (that will be the input for calculating the cost of successor operators). In the case of data materialization (despite materialization is not appropriate for such currency requirements) the inter-process delays with mediator interfaces, estimated in the worst case as the refreshment period, is the predominant delay.

Table 5.1 summarizes the calculation strategies for both factors.

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

**Table 5.1 – Calculation of property values for currency and timeliness factors**

When an activity has several predecessors, the maximum input freshness value is taken, i.e. the combination function returns the maximum. (See Chapter 3, Sub-section 3.1 for further details on combination functions).

The following properties were associated to the quality graph:

- RefreshPeriod (last operation node): Represents the difference of time between two refreshments of materialized data. It is defined during mapping generation.
- Capacity (operation node): Represents the quantity of tuples that the node can process per second. It is defined during mapping generation.
- Selectivity (operation node): Represents the percentage of input tuples (product of numbers of tuples of both input relations, in the case of a join) that constitutes the operation result. It is defined during mapping generation.
- Tuples (source and operation nodes): Number of tuples delivered by a node. For source nodes, it is provided by source administrators. For operation nodes, it is calculated from the number of tuples of input nodes and operation selectivity.
- UpdatePeriod (source nodes): Represents the difference of time between two updates at a source. It is provided by source administrators.

Several freshness evaluation algorithms where instantiated, overloading the *getSourceActualFreshness*, *getInterProcessDelay*, *getProcessingCost* and *combineActualFreshness* functions according to the DIS properties discussed previously. Concretely, we provided the following algorithms:

- *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 considered is the inter-process delay caused by data materialization.

The pseudocodes of such functions are detailed in Annex B. Next sub-section describes the experimentations with this application.

### 3.1.4. Experimentation

As we previously motivated, the experimentation with this applications has two main goals: (i) instantiate the framework and evaluate data freshness in this concrete application scenario in order to validate our evaluation approach, and (ii) test different evaluation algorithms adapted to different DIS properties and user expectations.

To achieve these goals, we implemented several freshness evaluation algorithms, as detailed in previous sub-section, and we simulated different scenarios by changing user preferences. Source property values (Tuples and UpdatePeriod) were manually set during test configuration. DIS property values (RefreshPeriod, Capacity and Selectivity) were randomly generated during query generation. All these properties, as well as the quality graphs representing mediation queries, were read from the metabase.

We executed the evaluation algorithms and used the graphical functionalities of DQE for analyzing evaluation results. Two main functionalities were particularly used: the coloring of graphs not achieving freshness expectations and the coloring of critical paths. This allowed simulating DIS reengineering analysis. Furthermore, some property values were manually changed during the simulation (using the Group/Property panel of DQE), for example, the refreshment frequency or the capacity of a node. Then, the evaluation algorithms were re-executed with the new property values, allowing the simulation of some basic kinds of *what-if* analysis.

The evaluation results, i.e. labels of the quality graphs, were persisted in the Metabase and thereafter used by the *User Profile Manager* in order to select the mediation query that better adapts to user preferences. The selection principle was very simple: mediation queries were ordered according to data freshness and other criteria, and the one providing the best quality (in a multi-criteria aggregation) was selected.

In order to provide other quality criteria, we implemented simple quality evaluation algorithms for computing *response time* and *confidence* estimations. Response time was computed adding the processing cost property calculated for data freshness evaluation. Confidence was computed as an average of the source confidence values, set as properties of source nodes. Despite the simplicity of the test, this allowed to validate the possibility of extending the framework (and the tool) for the evaluation of other quality factors.

As results of our experimentations we obtained some conclusions. Firstly, the application of the instantiation method to a given scenario is straight forward. In addition, we found that the knowledge about many properties and its estimation can be reused for other scenarios. The implementation of the overloaded functions is also very simple and its integration to the framework is straight forward. Finally, the graphical functionalities of the tool allowed an easy interpretation of evaluation results, especially, the visualization of critical paths is very useful for targeting enforcement analysis.

## 3.2. Evaluating data freshness in a web warehousing application

Web warehousing (WW) systems extract and integrate data from a set of conceptually-related web pages and store it in a Data Warehouse in order to allow decision making. WW extraction processes have to solve many problems related to the heterogeneity of web pages and the autonomy of web sources, including finding new relevant pages, detecting changes in the pages, accessing pages with different formats and transforming data to a common format. As many sources can provide the same information, extracted data must be reconciled in order to provide consistent and not duplicated information.

In this sub-section we study data freshness evaluation in the context of a WW application. Concretely, we instantiate the freshness evaluation algorithm to a WW scenario and we use the evaluation results in the integration process in order to return the freshest data to the user.

The main goal of this experiment is the practical use of our approach in a real application. This implies the modeling of several types of processes with different synchronization policies and different assessment methods for property values. The challenge is to show that our framework allows the easy representation of the application and that the freshness evaluation algorithm can be easily instantiated for this scenario.

### 3.2.1. Description of the application

In this sub-section we describe the Web Warehousing architecture proposed in [Marotta+2001], which is based on two types of modules: *wrappers* and *mediators*. The goal of a wrapper is to access a source, extract the relevant data, and present such data in a specified format. The role of a mediator is to merge data produced by different wrappers or mediators. Figure 5.7 shows an overview of the Web Warehousing architecture, starting with data extraction from Web documents and finishing with user interfaces or applications querying the system.

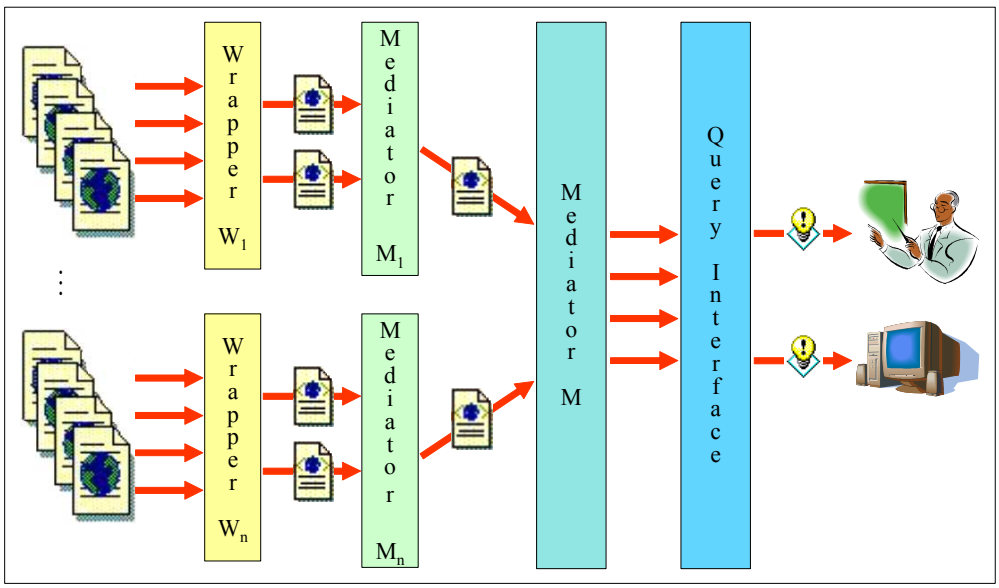


**Figure 5.7 – Overview of the Web Warehousing architecture**

The set of Web pages is grouped according to the information to be extracted from them. For each group of pages, a request schema and an associated wrapper are defined. The wrapper extracts the data in the pages and structures it according to the request schema. The output of each wrapper is a set of identical schemas populated with the corresponding data of each page. The wrappers also detect changes in Web pages.

A global integrated schema is also defined, which represents a unified view of the data manipulated by the system. As a direct consequence, data integration follows the local as view approach, where each data source is defined as a view of the global schema. The proposed architecture contains two kinds of mediators: (i) those integrating the results returned by each wrapper (called $M_i$) which only perform instance mappings since input schemas are identical, and (ii) a global mediator integrating the data returned by the $M_i$ mediators (called $M$) which performs schema and instance mappings.

We briefly describe each module; a larger description can be found in [Giaudrone+2005] [Marotta+2001].

- *Wrappers*: Information search is oriented by a request schema (expressed in an XML schema). The information request is build using all concepts of the request schema, incorporating synonyms of the concepts (e.g "movie" and "film") and instances of the concepts (e.g. "movie" and "Harry Potter") obtained from a domain ontology. It is used for identifying relevant Web pages and retrieving data from them. The retrieval algorithm is based on page structure; it evaluates rules for each element of the information request in order to locate it in the page. The result is an XML document for each relevant page, formatted according to the request schema.
- *$M_i$ mediators*: An $M_i$ mediator integrates the XML documents returned by wrappers, solving possible data conflicts. When a same element is found in several input pages, conflicting attributes are solved using a source-trust policy (the value from the source with the greater trust is chosen), non-conflicting attributes are taken from its source. The integration process can be extended to handle other conflict-resolution policies. The mediator also uses the domain ontology for unifying terms (e.g. "United States", "EE.UU" and "USA") and performs some basic cleaning for refusing invalid data elements (e.g. those containing null values for mandatory attributes). The result is an XML document that contains all extracted information for a request schema.
- *M mediator*: The M mediator integrates the data produced by $M_i$ mediators, also solving structural conflicts. The final result is a relational DW. The DW schema is designed by applying a set of high-level transformations to the request schemas [Marotta 2000]. Each transformation takes as input a set of relations and produces as output a set of relations, corresponding to high-level operations such as aggregations, denormalizations, calculations, etc. Data conflicts are solved following the approach of [Calvanesse+1999], which declaratively specifies suitable matching and reconciliation operations.

Regarding synchronization policies, some sources have the capability of announcing changes to wrappers (push policy). After being notified of a change in a page and having verified that it is a significant change (not only a format change) the wrapper executes. For the other sources, the wrappers periodically compare pages with their previously recorded versions (pull policy). Mediators follow push policies, which means that they execute when a previous module produced a new output (an XML document, input for the mediator). It is possible that changes are not reflected in the mediator output, for example, if the new data was duplicated in other source, or if it is conflicting with data coming from a source with higher priority. All modules materialize data, either in XML documents or in a relational database. User queries follow pull policies. Other application scenarios with different synchronization policies where studied in [Peralta 2006].

The following case of study (taken from [Giaudrone+2005]) illustrates the approach along the sub-section:

> **Example 5.2.** Consider a web-warehousing application that extracts web data about cinema. The application has two wrappers, the one extracting information about movies (title, original title, country, year, duration, genre, etc.) and the other extracting information about actors (name, nationality, place of birth, etc.). Figure 5.8 shows an example of a web page with information about a movie, showing some attributes of the request schema that where recognized by the wrapper in the page text. Detailed information about the request schemas, as well as the ontologies describing such concepts, can be found in [Giaudrone+2005].
>
> Two mediators: $M_1$ and $M_2$ take as input the XML documents with the extracted information and produce two XML documents integrating extracted data. Mediator M transforms such documents to the relational model, integrates and transforms data, populating five materialized views that conform the DW. Users access the DW data through view interfaces, which allows queries on *producers*, *movies*, *plays, statistics on plays* and *actors*. □


| Attribute | Description |
|---|---|
| Título | Title (Spanish) |
| Título original | Original title (original language) |
| Paises | Production countries |
| Año | Opening year |
| Duración | Duration in minutes |
| Genero | Artistic genre |
| Productora | Production company |
| Audiencia | Number of tickets sold |
| Idioma | Original language |
| Recaudación | Total takings |
| Actores | Names of the main actors |
| Directores | Names of the directors |
| Productores | Names of the producers |

**Figure 5.8 – Extracting information about an actor: (left) web page, (right) request schema**

The following sub-section shows how this case of study is modeled in DQE.

### 3.2.2. Modeling of the application in DQE

In this sub-section we illustrate the instantiation of the quality evaluation framework for the case of study introduced in previous sub-section. Concretely, we model quality graphs, describing the representation of the activities and their interaction. We consider a simplified version of the case of study, accessing to a small number of web pages ($\{mov_1, mov_2, mov_3\}$ for movies and $\{act_1, act_2, act_3, act_4\}$ for actors).

The quality graph modeling the application is shown in Figure 5.9. Wrappers execute for each page, generating an XML document with the extracted information. We represented several instances of wrappers (activities $w_{11}$, $w_{12}$, $w_{13}$, $w_{21}$, $w_{22}$, $w_{23}$ and $w_{24}$) to explicitly show the data flow, however, there is a unique process that is invocated for several pages. Mediators $M_1$ and $M_2$ take as input the XML documents with the extracted information and produce another XML document integrating all extracted data; they are represented by activities $m_1$ and $m_2$. Mediator M integrates information produced by mediators $M_1$ and $M_2$. Input data is transformed to the

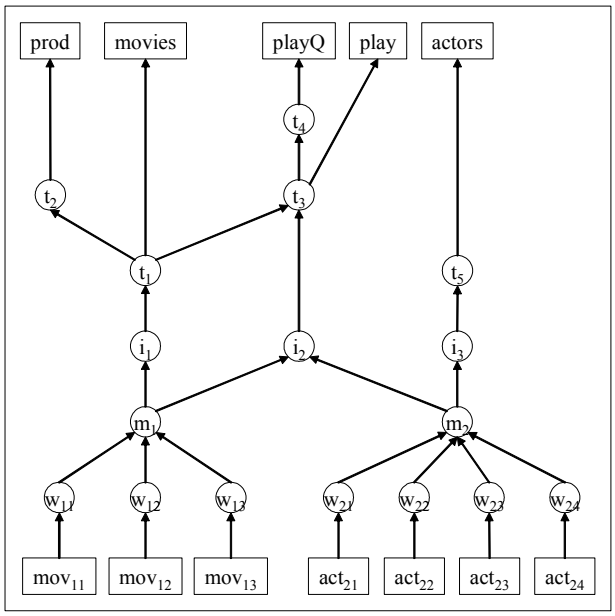


**Figure 5.9 – Quality graph for the web warehousing application**

relational model and stored in two ODS relations (activities $i_1$ and $i_3$). The DW is populated by a set of processes that transform ODS data, represented by activities $t_1$, $t_2$, $t_3$, $t_4$ and $t_5$. Users access the DW data through the view interfaces $v_1$, $v_2$, $v_3$, $v_4$ and $v_5$. Control flow coincides with data flow. The data types of the data produced by each node (e.g. XML documents, temporal relations, materialized views) are labels of the quality graph.

Let's analyze some characteristics of the scenario:

- Users are interested in seeing "recent" information about films and actors (timeliness). However, system administrators (who monitor change detection and view refreshment) need currency measures.
- The nature of data is different in the various sources. Actors' information (name, nationality, etc.) is quite stable, some movie information (title, country, genre, etc.) has a relatively long-term change frequency but some attributes (audience, takings, etc.) frequently change.
- Concerning DIS implementation, the application materializes data. Furthermore, all activities materialize data, either in XML files or in a relational database. Activities are simple automatic extraction, reconciliation and aggregation processes.
- There are different synchronization policies: synchronous push and asynchronous pull between sources and wrappers, synchronous pull between view interfaces and targets, asynchronous pull between transformations and view interfaces, and asynchronous push among the other activities.

The scenario is characterized as follows:

- **WW scenario:**
  - Freshness factor: currency and timeliness
  - Nature of data: stable, long-term-changing and frequently-changing data
  - Architectural techniques: materialization techniques
  - Synchronization policies: push-push, pull-pull and pull/pull policies

The characteristics of the scenario are taken into account to determine the relevant properties and estimate them, which is discussed in next sub-section.

### 3.2.3. Estimation of property values

Users expect timeliness values of "two weeks" for movies, play, statistics and producers' information and of "six months" for actors' information. System administrators tolerate currency values of "a week" for all user interfaces. With such freshness requirements, the "day" is a good unit for measuring freshness and properties values. Properties with values ranging in "some minutes" or less can be neglected. Activities processing costs are negligible compared with freshness expectations. However, due to data materialization, inter-process delays are relevant. In addition, as many sources may rarely update data, source data actual timeliness is also relevant.

We can now estimate the values of relevant properties, i.e. inter-process delays and source data actual freshness.

Data currency is measured as the time passed since data extraction, so it is zero at extraction time (source data actual currency). Data timeliness is measured as the time passed since page update[*]. The estimation of update time depends on the synchronization policy between sources and wrappers. For wrappers with push policy the notification time is used. Wrappers with push policy extract data immediately after being notified of a change, so source data actual timeliness is negligible for them. For wrappers with pull policy, source data actual timeliness is more difficult to estimate. When a change is detected (comparing with the previous version) the time passed between two consecutive pulls (pull period) is a worst case estimation. When no change is detected, the time passed since the last change detection should be added. More precise estimations may be obtained from knowledge about sources (e.g. if we know that a source is updated every Monday) or using statistical techniques (e.g. frequently accessing a source during some periods to determine source update behaviors). Some sources allow the access to file system metadata (as file modification time) or include the "last update time" inside the page text. Our tests have shown that none of these indicators are reliable enough. The former depends on system configuration (regional settings and correct configuration of local time) and the latter is not always updated when modifying a page.

[*] Depending on user needs, display changes can be ignored.

Due to push policies, activities having a unique predecessor (which is the case of $i_1$, $i_2$, $t_1$, $t_2$, $t_4$ and $t_5$) always execute after their predecessor, so there is no inter-process delay among them. However, an activity having several predecessors (which is the case of $m_1$, $m_2$ and $t_3$) executes when an input activity produces new data but may read data materialized several days ago by the other input activities. The inter-process delay among them is measured as the difference of time between their executions. Keeping statistics of execution dates the precise calculation is straightforward. Average values and upper bounds can be also obtained from statistics or can be deduced from source update policies*. Note that as some modules may not produce output data (e.g. if the changes are not relevant or where provided by other sources), successor activities do not need to execute because they would produce the same output too. Regarding data, the output of successor modules will be identical to the existing one, nevertheless, regarding data quality, the output will be fresher because it is built with more recent data (the page was updated). So, the execution date of all successor activities is updated even if they do not execute. The time passed between the materialization of data and the moment a user poses a query, determines the inter-process delay between transformations and view interfaces. Precise values, average values and upper bounds can be also obtained from statistics of execution dates.

Table 5.2 summarizes the calculation strategies.

| | **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* |

**Table 5.2 – Calculation of property values with different types of estimation**

Remember that when an activity has several predecessors, the freshness of data coming from them is combined, consolidating an input freshness value for the activity and that the combination strategy depends on the activity semantics. In our case of study we take into account data volatility (movie information is more volatile than actors' information); the strategy consists of ignoring input values of sources providing more stable information. For example, $t_3$ will return the input value from $t_1$. The nature of data dimension is used to define this strategy. When input activities have the same data volatility (which is the case of wrappers), the combination function performs a weighted average, where weights are data volumes (number of movies/actors). Table 5.3 summarizes the calculation strategies.

| | **Precise value** | **Average case** | **Worst case** |
|---|---|---|---|
| **When different data volatility** | Input value of most volatile input | | |
| **When 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 5.3 – Calculation of combination functions with different types of estimation**

In next sub-section we discuss the instantiation of freshness evaluation algorithms taking into account these properties.

* Despite source autonomy, in some application domains it is common to have information about source update policies. For example, in the cinema domain, most timetables pages are updated weekly (e.g. at Wednesday) because cinema schedules generally change weekly.

### 3.2.4. Data freshness evaluation

The freshness evaluation algorithm is instantiated overloading the *getSourceActualFreshness*, *getInterProcessDelay*, *getProcessingCost* and *combineActualFreshness* functions according to the DIS properties that are most relevant for the scenario. Pseudocodes of such functions are detailed in Annex 2.

The following properties were associated to the quality graph:

- AnnounceChanges: Represent the capability of the source to announce changes to the corresponding wrapper. It is defined by DIS administrators and set as propagate value (at DIS design time).
- PullPeriod (wrapper nodes): Represents the difference of time between two data extractions. It is defined by DIS administrators and set as propagate value (at DIS design time).
- DataVolatility (data edge): Represents the volatility of data flowing in the edge. Expert users provide the volatility values for wrappers, which are replicated to data edges and set as propagate values (at DIS design time).

The following logs were used to register more volatile properties:

- Execution-ChangeDetection (source, wrapper): At each execution, a wrapper registers two values: (i) its execution time and (ii) the previous change detection time. The log provides statistics of the difference between such values, i.e. the last, average and maximum entry.
- Execution-Execution (activity, activity): At each execution, a mediator task or view interface registers two values for each predecessor module: (i) the execution time of the predecessor (obtained from the output of the module or a previous entry of this log) and (ii) its execution time. The log provides statistics of the difference between such values, i.e. the last, average and maximum entry.
- Execution-ChangeDetection (source, wrapper): At each execution, a wrapper or mediator task registers one value: the number of produced tuples/elements. The log provides statistics on such values, i.e. the last, average and maximum entry.

Statistics on execution times and change detection times where recorded by wrappers and mediators [Vila+2006]. Statistics on user queries have been recorded by triggers on view interfaces (we simulated random accesses in order to test the proposal).

Next sub-section describes the experimentations with this application.

### 3.2.5. Experimentation

In order to test the practical use of our approach in a real application, we modeled the WW application as a quality graph and we analyzed its properties. The main difficulty resided in the modeling of different update propagation policies, which caused different ways of determining inter-process delays. The assessment of source data freshness was an additional challenge. Several property values (announcement policies, pull periods and data volatility) were determined at design time and set as labels of the quality graph. However, other property values (amounts of time among execution of activities) are more volatile and needed fresh statistical estimations in order to assess property values. We measured such property values from logs recorded by wrappers and mediators. Such modules were explicitly modified in order to record such statistics; details on the implementation of logs can be found in [Vila+2006].

A first conclusion of our experimentation is that, even the DIS and its relevant properties can be easily modeled in DQE, the assessment of source data quality and DIS property values can be a tedious task and may imply the development of specialized routines. In this thesis we do not deal with source quality and property assessment, but we found that the analysis of assessment techniques can be an interesting line for future works. This topic is discussed in Chapter 6.

Concerning the implementation and execution of evaluation algorithms, we confirm the conclusions obtained in previous experiment, i.e. the application of the instantiation method to a given scenario is direct, the implementation of the overloaded functions is very simple and its integration to the framework is straight forward.

An indirect result of the experimentation was the use of quality values to improve the data integration process. Concretely, the $M_i$ mediators where modified in order to take as input the freshness of data extracted by wrappers. Data quality is introduced in the integration process in two ways:

- *Data filtering*: The sources do not achieving a certain quality level are discarded. To this end, each mediator has associated a quality bound (a freshness threshold) and does not process inputs (XML documents corresponding to Web pages) which freshness value is greater than the threshold.
- *Reconciliation policy*: The source with the higher quality is selected when there are conflicts. To this end, a new reconciliation policy was defined for the WW application, based on the freshness of conflictive data, i.e. the data coming from the freshest input (XML document corresponding to a Web page) is chosen.

Clearly, the new implementations of mediators produce fresher data than their previous versions. However, as data freshness is not the unique criterion that should be taken into account in data reconciliation, the real improvement has not been tested. We will discuss it as perspective.

## 3.3. Evaluating data accuracy in a data warehousing application

The loading and refreshment of a Data Warehouse (DW) involves several types of activities, ranging from the extraction of data from several sources, the transformation of this data and its loading into the DW. Such processes is generally know as ETL (*Extraction, Transformation and Loading*) processes. The transformations applied to extracted data consist, basically, of data cleaning and formatting routines. Data cleaning is necessary to assure the quality of DW data. It involves, among others, the correction of errors, the elimination of redundancy and the resolution of inconsistencies, as well as the achievement of the business rules. Formatting serves to structure data according to DW requirements. It includes the adjustment of data to DW schema, the changing of data format and the aggregation of data.

In this section we analyze the ETL processes of a concrete DW application, which manages information about students of a university [Etcheverry+2005].

The goal of this experiment is threefold. Firstly, we want to model a big complex application in DQE, having many activities and varied properties. Secondly, we intend to experience with different accuracy metrics (for different data types and error types) and with varied measurement techniques. Finally, we want to validate the accuracy evaluation approach in a real application.

### 3.3.1. Description of the application

We limited our analysis to the ETL processes of a data mart of the university DW. The data mart consists of two multidimensional fact tables and eleven dimensions, which describe university students, the courses they followed, the exams they took and their marks.

ETL processes where studied at three levels of abstraction: The highest level provides a macro vision of ETL packages and their precedence relationships. There are 3 packages, one for loading dimensions and one for loading each fact table. The medium level describes the dataflow inside each module, identifying ETL routines (transactions). The lowest level describes the sequence of operations that implement each ETL routine. Each ETL module contains between 5 and 20 routines, each one containing up to 10 operations. Figure 5.10 illustrates the routines that compose the loading of one of the fact tables (medium level). A complete description of ETL modules, at the three abstraction levels, can be found in [Etcheverry+2005].

The modeling of the application in DQE is straight forward. We defined a hierarchy of quality graphs, with three levels, representing activities at the three abstraction levels. The highest-level graph allows visualizing the whole process; activity nodes represent ETL packages. The medium-level graph represents the dataflow among ETL routines and the lowest-level graph shows ETL operations. Source nodes represent source relations and target nodes represent dimension and fact tables. See Sub-section 4.1.1 of Chapter 3 for details on modeling quality graphs at different abstraction levels.

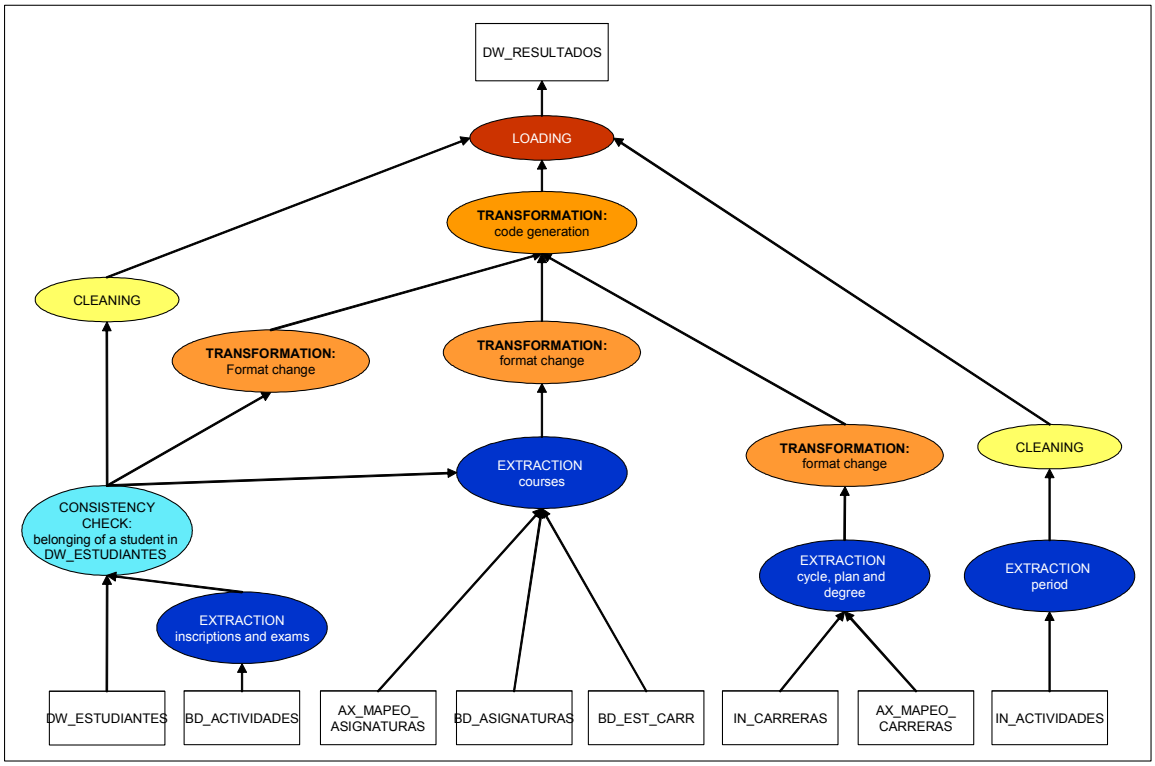


**Figure 5.10 – Routines of an ETL package**: extraction, consistency check, transformation, cleaning, and loading.

### 3.3.2. Accuracy assessment

In order to measure accuracy of source data, we experienced several metrics and measurement functions. The implementation of the assessment techniques was carried out by two master students: Lorena Etcheverry and Salvador Tercia [Etcheverry+2006], the former having worked in the development of the DW application some years ago. They implemented specialized techniques for detecting and measuring four types of errors*, which are summarized in Table 5.4.

The assessment of semantic correctness was simulated in order to avoid comparisons with real-world. Source relations were polluted in order to represent semantic errors. Copies of source relations (before pollution) were used as referential tables representing real world. The assessment method (*CHECK_REF* function) consisted in the comparison of source data with referential data. This pollution strategy allowed the implementation of automatic assessment methods.

An assessment function (*CHECK_RULE*) was implemented for measuring syntactic errors. The function verifies if data satisfies a given format, given by a range or a grammar. Several attributes already contained errors but further errors were simulated either polluting source data or defining restricted formats. An example of the latter was the definition of a grammar, consisting of 7 digits, for telephone numbers. As in Uruguay home telephones have 7 digits and mobile telephones have 9 digits, mobile telephones are detected as having an illegal format.

We detected several cases of imprecision in source data. For example, most instances of *city* attribute (DW_ESTUDIANTES table) correspond to cities, but we also found country names and continent names. We defined a referential table containing the instances of the *city* attribute and assigning them a level of precision. Specifically, precision is 1 for city names and takes inferior values for country and continent names. The assessment function (*CHECK_LEVEL*) obtains the precision measure from the referential table. Implementation details can be found in [Etcheverry+2006].

* The work also included the assessment of data consistency, considering further types of errors, including the presence of null values for mandatory attributes and the violation of attribute dependencies, uniqueness constraints and referential integrity constraints. As the analysis of data consistency is out of the scope of this thesis, we do not provide details on its measurement.

| Accuracy factor | Error description | Assessment method |
|---|---|---|
| Semantic correctness | Values that not correspond to a real-world entities. | *CHECK_REF*: Compares data against a referential table representing real-world. |
| Syntactic correctness | Out or range values | *CHECK_RULE*: Evaluates if data satisfies a range rule (expressed as a numeric interval) |
| | Illegal format | *CHECK_RULE*: Evaluates if data satisfies a format rule (expressed by a grammar) |
| Precision | Insufficient precision | *CHECK_LEVEL*: Compares data precision against a referential table |

**Table 5.4 – Typical errors and their corresponding assessment functions**

### 3.3.3. Experimentation

In order to test our accuracy evaluation algorithm we defined an application scenario inspired on the DW loading processes. We evaluate accuracy of data loaded from DW sources, i.e. data stored in the DW. As our evaluation approach does not considers data materialization and only manages JSP queries, ETL processes were simplified to JSP queries, ignoring data cleaning, consistency checks and complex transformations. The simplified loading queries allowed the validation of our approach with a representative set of queries.

Source relations were partitioned based on accuracy measures (using the assessment functions explained in previous sub-section). Several partitioning criteria were used. In some cases we used knowledge about error distribution (obtained executing test queries and keeping statistics); in other cases we designed pollution methods in order to obtain certain error distribution (e.g. with defined a set of areas and we polluted them with different error ratios). It was not necessary, for our tests, to use Rakov's partitioning algorithm. The loading queries were rewritten in terms of partitions. The selectivity of each rewriting was estimated using simple statistics.

This test allowed the validation of our measurement approach with real data. We experimented with different accuracy metrics and assessment functions as well as with different partitioning criteria. We can affirm that the approach is viable and obtained accuracy values may approximate the actual values obtained assessing accuracy of exact query result. We plan to make additional tests in order to measure the precision of the obtained accuracy values. In Chapter 4 we showed that the evaluation approach depends on the techniques used for measuring accuracy of source relations, partitioning them and estimating query selectivity. We aim to quantify such influence by testing different techniques and their impact in the partitioning of query results. By performing such tests, we can compare results with those obtained applying the Naumann's approach [Naumann+1999] (which considers uniform distribution of errors) and the Rakov's approach [Rakov 1998] (which is build knowing the exact query result). Both approaches were described in Sub-section 3.1 of Chapter 4. We defer this test to near future.

As future work, we hope to extend the approach to consider other types of activities, specially, those correcting errors (e.g. data cleaning and format standardization routines). This DW application can be taken as a case of study for proposing evaluation techniques and algorithms for this type of application scenarios. The measurement approach proposed in Chapter 4 is a first step towards the definition of general evaluation methods that might be instantiated to several types of application scenarios. This topic is discussed as perspective in Chapter 6.

Another result of this experiment, was the use of DQE for modeling a big and complex DIS, with many activities and varied properties (materialization, complex transformations, etc.).

## 4. Evaluation of Performance and Limitations of the DQE Tool

In this section we present the results of performance and limitations tests performed with the DQE tool.

In order to evaluate performance and limitations, we generated some test data sets (consisting in a set of quality graphs, data sources, data targets and property values) and we executed an evaluation algorithm (the *ActualFreshnessPropagation* algorithm) on each graph. The second version of DQE was used for the tests, so quality graphs are loaded from the metabase and stored in memory. Thus, one of the objectives of the tests was

to determine the limitations of the tool, i.e. the number of quality graphs that can be loaded in a session being the input for the execution of quality evaluation algorithms. We made three types of tests:

- *Application limits*: The test consisted in generating large data sets in order to know the number of graphs (with different topologies) supported concurrently by the tool.
- *Quality evaluation performance*: The test consisted in measuring the time it takes the tool to evaluate data quality on a set of graphs, i.e. executing the evaluation algorithm on the graphs.
- *Loading performance*: The test consisted in measuring the time it takes the tool to communicate with the metabase for loading the set of graphs and their associated metadata.

The obtained results allow affirming that the tool can be used for large applications (modeling hundreds of graphs with hundreds of nodes each). Scalability (using the last DQE version) is discussed as perspective in Chapter 6. The following sub-sections discuss the generation of test data sets, describe the tests and present the results.

## 4.1. Generation of test data sets

In this section we describe the data sets used for the various tests, the process used for generating them and the results of the generation.

We generated three types of test data sets:

- *Small data sets*: Fifty data sets where generated varying the number of graphs (from 10 to 100) and the number of nodes of each graph from (10 to 50).
- *Bulk data sets*: Fifty data sets where generated varying the number of graphs (from 100 to 5000); graphs are small (15 nodes).
- *Big data sets*: A hundred data sets where generated varying the number of graphs (from 100 to 1000) and the number of nodes of each graph from (100 to 1000).

The main purpose of generating small data sets is validating the correctness of the generator (checking if generated graphs are well-formed, with varied topologies and adequate property values, allowing the correct execution of the evaluation algorithm). In addition, as most application scenarios are represented with small graphs, we also are interested in evaluating the tool performance. The purpose of generating bulk and big data sets is to know the maximum number of small/big graphs supported in memory and the performance of the quality evaluation algorithms with such data sets.

Next sub-section describes the generation process.

### 4.1.1. Generator of data sets

We implemented a generator of data sets, which works in three phases: (i) generates a set of sources and a set of targets, (ii) generates a set of quality graphs, and (iii) associates property values to the nodes and edges of the graphs.

The first phase is straightforward. Sources and targets are sequentially named (e.g. S1 and T1).

In the second phase, a set of graphs is also generated, with sequential names (e.g. graph1). Nodes and edges are generated in six steps, as illustrated in Figure 5.11 (all edges are mixed data edges, representing data and control flow). The generation steps are the following:

1. A set of source nodes is randomly selected from the source set and a set of target nodes is randomly selected from the target set. In Figure 5.11a, three sources and two targets are selected.
2. A set of activity nodes is generated, sequentially named (e.g. A1). In Figure 5.11b, six activities are generated.
3. A set of edges between source and activity nodes is generated in the following way: for each source node, an activity node is randomly selected (a different activity for each source node) and an edge between them is added. In Figure 5.11c, activities A6, A2 and A1 are selected.
4. A set of edges between activity nodes is generated in order to link each isolated activity to the ones already connected (initially, those linked to source nodes). To do so, a set of activity nodes are randomly

selected from those already connected (which will be the predecessors) and an edge between each predecessor and the isolated activity is added. In Figure 5.11d, A6 and A2 are selected as predecessors for A3; then A5 is linked to A6 and A4 is linked to A3.

5. A set of edges between activity and target nodes is generated, in the following way: for each target node, an activity node is randomly selected (a different activity for each source node) and an edge between them is added. Activities with no successors have priority to be selected. In Figure 5.11e, activities A5 and A4 are selected.
6. If there are activities with no successors, a set of edges is generated for connecting them to other nodes. In order to avoid cycles, candidate nodes are those that are predecessors of some target node. In Figure 5.11f, an edge between A1 and A4 is added.

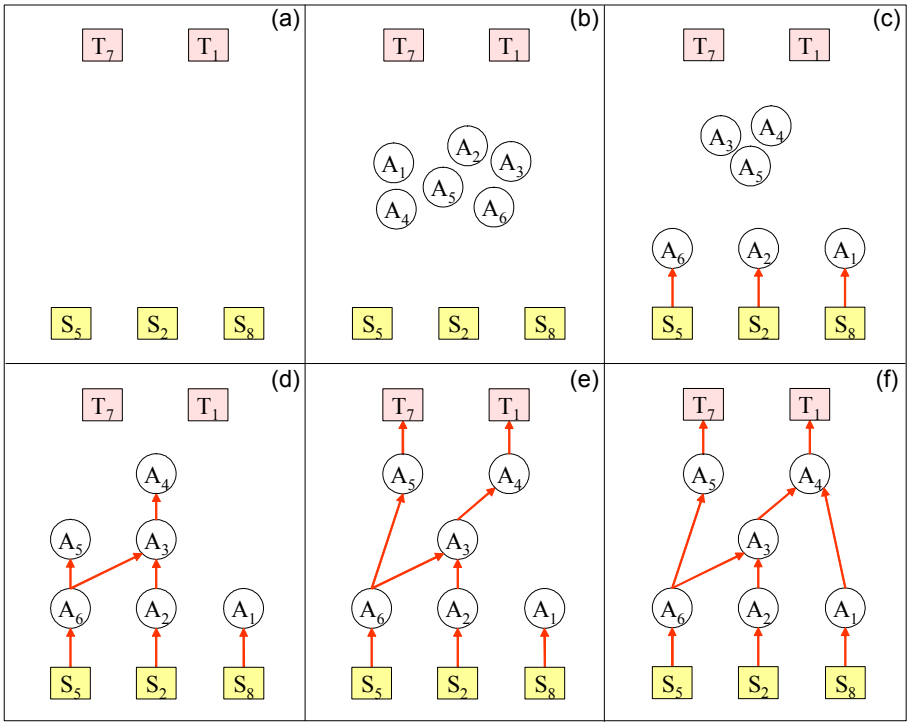


**Figure 5.11 – Generation of graphs**

In the third phase, a set of property values are associated to nodes and edges, in the following way:

- A source data actual freshness value is associated to each source, and then, to all source nodes (of the graphs) corresponding to the source.
- A target data expected freshness value is associated to each target, and then, to all target nodes (of the graphs) corresponding to the target.
- A processing cost value is associated to each activity node of the graphs.
- An inter-process delay value is associated to each edge of the graphs.

Table 5.5 lists the generation parameters and their default values. Four categories of parameters are defined: session components, number of graph nodes, input degree of activity nodes and property values. All these parameters are written in a configuration file.

Table 5.6 summarizes the parameters used for generating the three types of data sets.

The generator was implemented in Java (JDK 1.4) and stores the generated data set in an Oracle® database (Oracle 10g Enterprise Edition Release 10.1.0.2.0). In next sub-section we discuss a preliminary test performed for verifying that the generated graphs are well-formed and allow the execution of quality evaluation algorithms.

| Parameter | Description | Default |
|---|---|---|
| ***Session components***. Sources, targets and quality graphs are generated; properties and algorithms are fixed. | | |
| S | *Number of sources* in the session (integer number) | |
| T | *Number of targets* in the session (integer number) | |
| G | *Number of graphs* in the session (integer number) | |
| ***Number of graph nodes***. As nodes are of three types (source nodes, activity nodes and target nodes), three parameters are defined, one for each type. In order to generate heterogeneous graphs, the number of nodes is randomly set, given the intervals (minimum and maximum values) for the random selection. Some integrity rules assures that minimum values are appropriate (smaller than maximum values and, in the case of activities, bigger than the number of sources and targets randomly generated). Parameters are: | | |
| $\mathbf{N_s}$ | *Maximum number of source nodes* of the graphs (integer number) | |
| $\mathbf{n_s}$ | *Minimum number of source nodes* of the graphs (integer number) | $N_s/2$ |
| $\mathbf{N_t}$ | *Maximum number of target nodes* of the graphs (integer number) | |
| $\mathbf{n_t}$ | *Minimum number of target nodes* of the graphs (integer number) | $N_t/2$ |
| $\mathbf{N_a}$ | *Maximum number of activity nodes* of the graphs (integer number). | |
| $\mathbf{n_a}$ | *Maximum number of activity nodes* of the graphs (integer number). | $N_a/2$ |
| ***Input degree of activity nodes***, i.e. the number of edges incoming activity nodes (excepting wrappers, which have degree 1). In order to generate heterogeneous nodes, the input degree is randomly generated given the intervals for the random selection, namely: | | |
| **d** | *Minimum degree* (integer number) | 1 |
| **D** | *Maximum degree* (integer number) | 3 |
| ***Property values*** for nodes and edges. In order to generate different values, they are randomly generated given the intervals for the random selection. Furthermore, as different properties may have different significant values, intervals are specified for each property, namely: | | |
| $\mathbf{p_a}$ | *Minimum actual freshness* of a source node (integer number) | 0 |
| $\mathbf{P_a}$ | *Maximum actual freshness* of a source node (integer number) | 24 |
| $\mathbf{p_e}$ | *Minimum expected freshness* of a target node (integer number) | 24 |
| $\mathbf{P_e}$ | *Maximum expected freshness* of a target node (integer number) | 48 |
| $\mathbf{p_c}$ | *Minimum processing cost* of an activity node (integer number) | 0 |
| $\mathbf{P_c}$ | *Maximum processing cost* of an activity node (integer number) | 2 |
| $\mathbf{p_d}$ | *Minimum inter-process delay* of an edge (integer number) | 0 |
| $\mathbf{P_d}$ | *Maximum inter-process delay* of an edge (integer number) | 4 |

**Table 5.5 – Generation parameters**

| Data set | $N=N_s+N_t+N_a$ | G | S | T | d | D |
|---|---|---|---|---|---|---|
| Small | from 10 to 50 | from 10 to 100 | 10 | 10 | 1 | 3 |
| Bulk | 15 | from 100 to 5000 | 10 | 10 | 1 | 2 |
| Big | from 100 to 1000 | from 100 to 1000 | 200 | 200 | 1 | 3 |

**Table 5.6 – Generation parameters for the data sets**

### 4.1.2. Generation correctness

The small data sets were analyzed in order to check whether well-formed graphs were generated and visualize them. Figure 5.12 shows some of the graphs generated in a test (with $10<N\leq20$) and Figure 5.13 shows a bigger graph generated in another test (with $40<N\leq50$). The generator takes no care of node positions but allows users to reposition nodes (drag and drop) in order better visualize them. Repositioning, even if possible, is not viable for huge graphs[*].

A first remark is that some graphs are not connected. We analyzed the possibility of checking connection during generation (and adding some edges to connect disconnected sub-graphs), but we discarded the idea because of the cost of such checking. Disconnection is not a problem for quality evaluation algorithms since data quality is propagated along existing data flow. So, we permitted the generation of disconnected sub-graphs. However, we

[*] Screen captures correspond to the graphical interface of DQE version 2, differing from that of the current version shown in Sub-section 2.3.

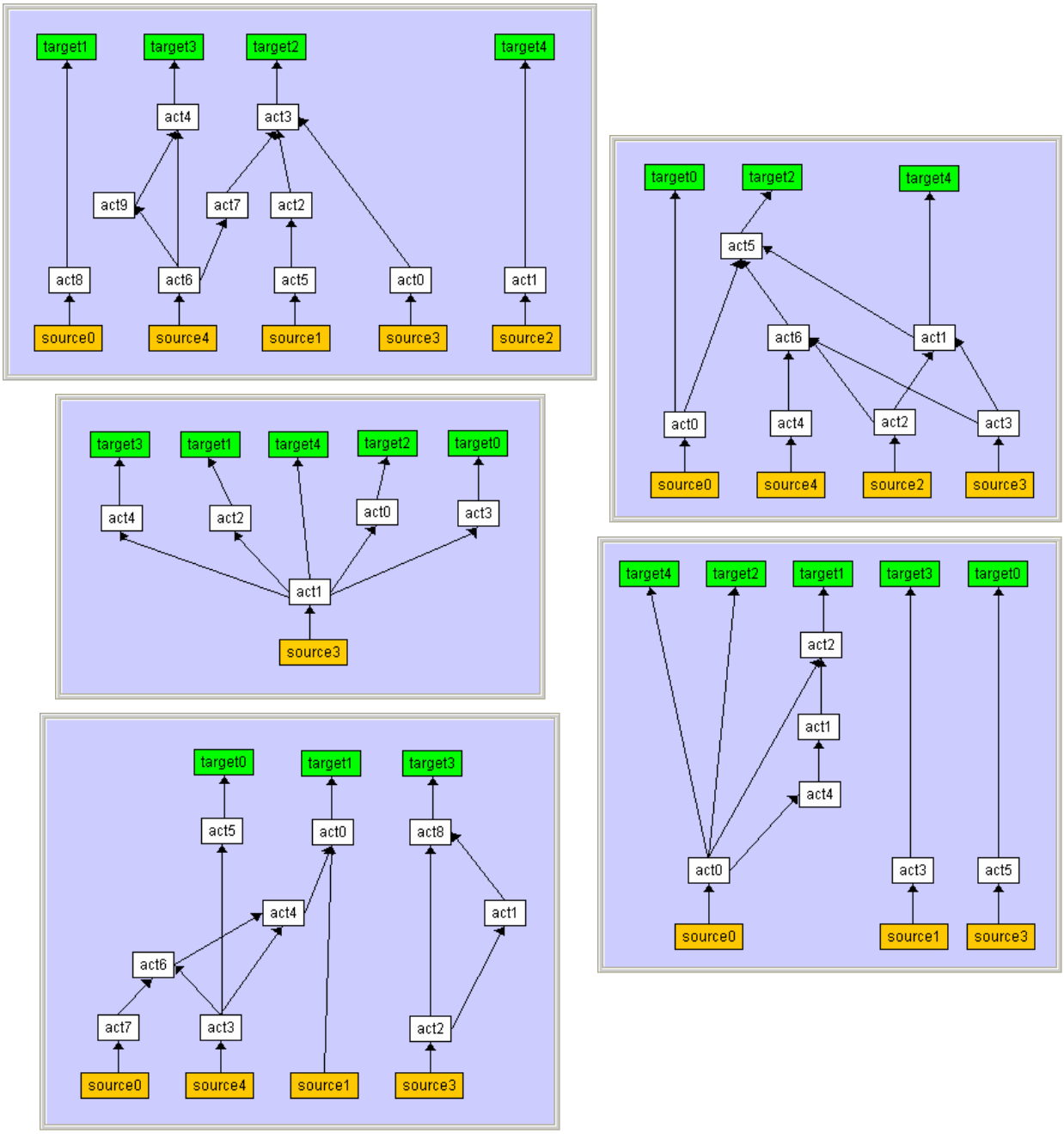


**Figure 5.12 – Some generated graphs having among 10 and 20 nodes**

assured that graphs respect the quality graph definition (Definition 3.4), i.e., graphs are acyclic, source nodes have no incoming edges and a unique outgoing edge, target nodes have a unique incoming edge and no outgoing edges. We also checked that all nodes and edges have the appropriated property values.

Property values were also verified, as well as the result of applying evaluation algorithms. The analysis of critical paths for the graph of Figure 5.13 is shown in Figure 5.14. Some big data sets were also analyzed (it was a tedious task). No figures are shown because graphs are too big to be displayed inside a screen.

A batch script, without graphical interface, was implemented to run the tests. It firstly invokes the generator indicating the name of a file containing generation parameters, it loads the generated session into the DQE tool, and it execute the *AcutalFreshnessPropagation* algorithm (presented in Chapter 3) for all generated graphs. Generation, loading and evaluation times (and other indicators) are logged in a file. All tests where repeated three times. In next sub-sections we discuss test results for application limits, quality evaluation performance and loading performance.

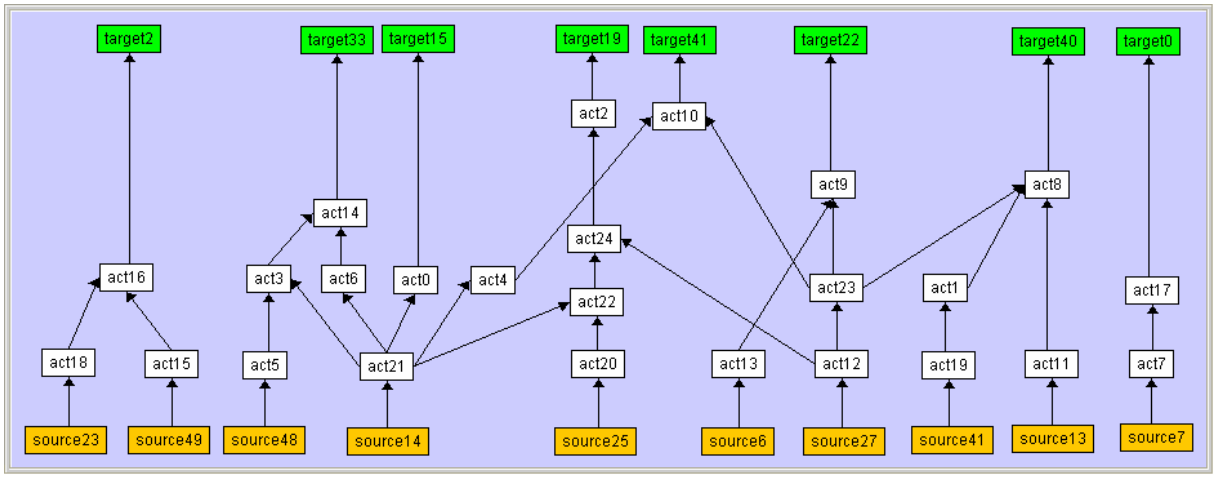


**Figure 5.13 – Generated graph having 9 target nodes, 25 activity nodes and 10 source nodes**

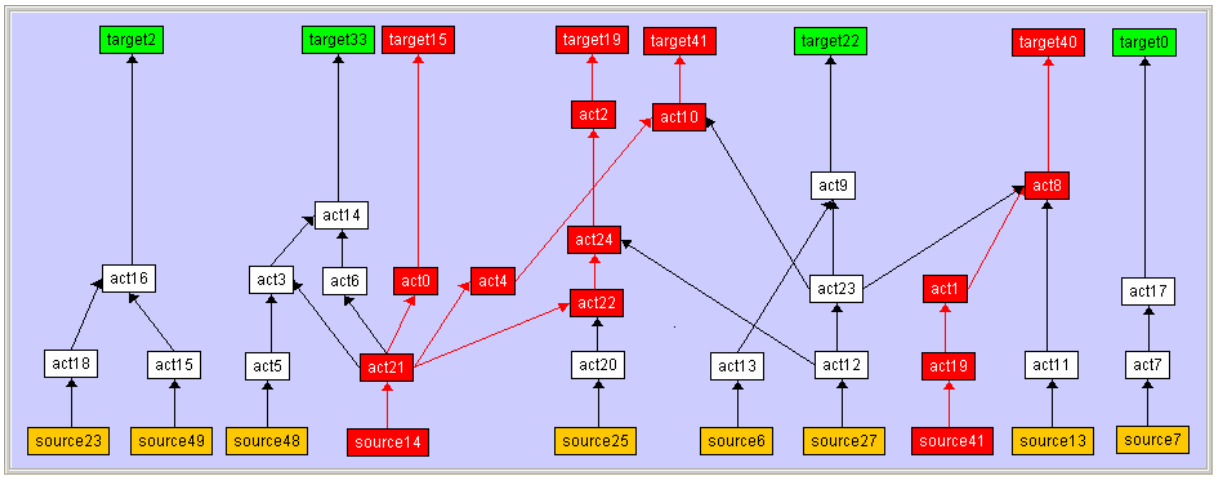


**Figure 5.14 – Highlighting critical paths**

## 4.2. Test of limitations

The bulk and big data sets where used to determine applications limits, i.e. know the number of graphs (with different topologies) supported concurrently by the tool. Remember that DQE version 2 loads data in memory, so, the test indicates how many graphs can be stored in memory for quality evaluation.

The result with the bulk data sets is that up to 2800 small graphs (15 nodes) can be treated in parallel; 2900 graphs can be loaded but memory overflow occurs during quality evaluation.

The result with the big data sets is shown in Table 5.7; colored cells indicates the data sets that were successful for all test repetitions, shadow cells indicate the data sets that were successful for all repetitions of the loading test but not for the evaluation test and uncolored cells indicates the data sets for which the tests failed for at least one repetition. Obviously, some data sets were not tested because of the fail of smaller data sets. The test shows that the tool supports data sets consisting in several hundreds graphs with several hundreds nodes. The tool can also support almost a thousand medium graphs (with a hundred nodes) or some huge graphs (with a thousand nodes). The application does not scale to thousands of huge graphs. Scalability must be provided replacing the memory storage by access to secondary storage, which is discussed as perspective in Chapter 6.

Next sub-section analyzes performance for the tests data sets that succeeded.

| N \ G | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 |
|---|---|---|---|---|---|---|---|---|---|---|
| 100 | | | | | | | | | | |
| 200 | | | | | | | | | | |
| 300 | | | | | | | | | | |
| 400 | | | | | | | | | | |
| 500 | | | | | | | | | | |
| 600 | | | | | | | | | | |
| 700 | | | | | | | | | | |
| 800 | | | | | | | | | | |
| 900 | | | | | | | | | | |
| 1000 | | | | | | | | | | |

**Table 5.7 – Application limits for the big data sets**

## 4.3. Test of performance

In this sub-section we analyze the tool performance for the three types of data sets. The main goal is to evaluate quality evaluation performance, but as quality graphs have to be loaded in memory in order to evaluate their quality, we also evaluate loading performance.

### 4.3.1. Quality evaluation performance

In this test we analyze quality evaluation performance. We measure the time it takes the tool to evaluate data freshness, i.e. executing the *ActualFreshnessPropagatiion* algorithm on all graphs of the data set.

The test with the small data sets shows that evaluation time is linear in the number of graphs and the number of nodes. This result is reasonable because the evaluation algorithm is applied to each graph, traversing each node once. Figure 5.15 shows the test result; the noise in the graphics is explained by the small magnitude of evaluation times (milliseconds), which makes difficult the filtering of other operation system routines that also consume time. The bigger data set (100 graphs of 50 nodes) was computed in 0.22 seconds.

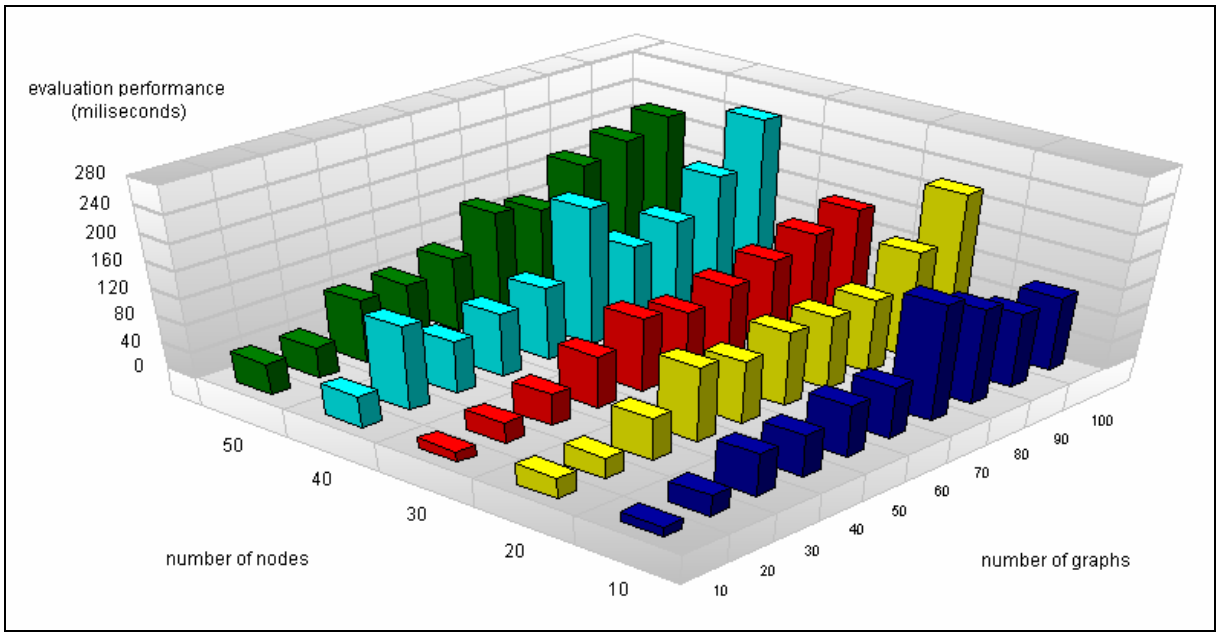


**Figure 5.15 – Evaluation times for the small data sets**

The test with the bulk data sets also shows that evaluation time is linear in the number of graphs. Figure 5.16 shows the test result. Quality evaluation can be performed in more than a thousand graphs in a second.

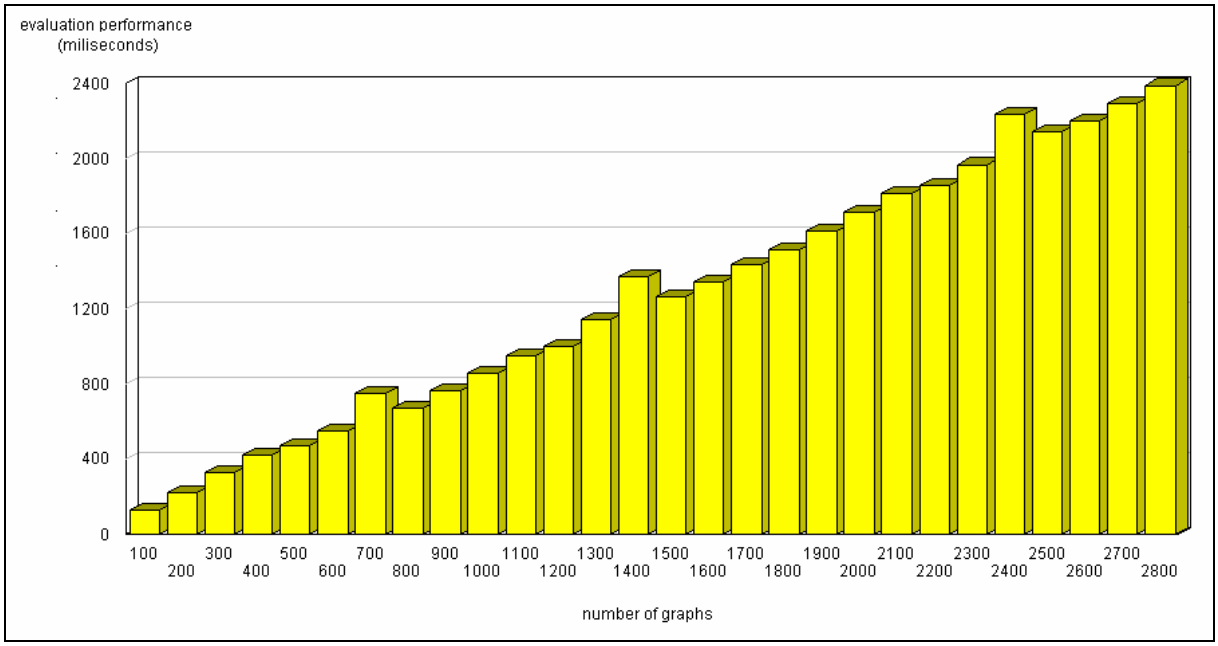


**Figure 5.16 – Evaluation times for the bulk data sets**

The test with the big data sets confirms that evaluation time is linear in the number of graphs and the number of nodes also for big graphs. However, the limit data sets (the biggest ones being supported) provoke a great number of memory faults, which cause a huge evaluation time, overdrawing the linear scale. Figure 5.17 show the test result. Quality evaluation in some hundred graph of some hundred nodes lasts some seconds.

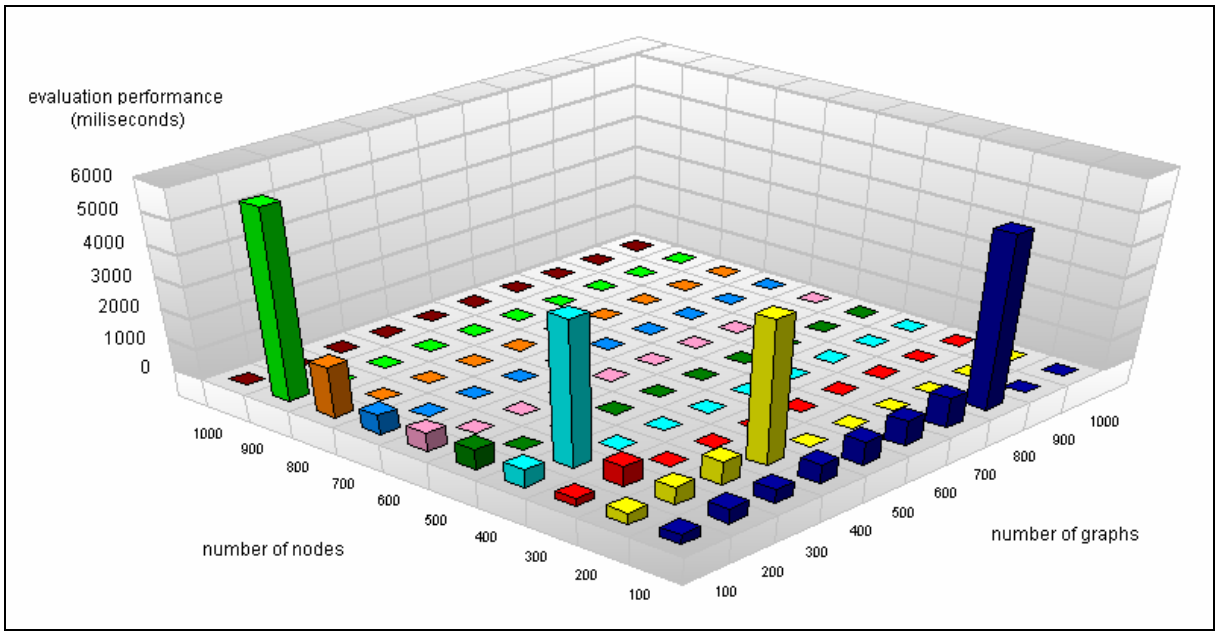


**Figure 5.17 – Evaluation times for the big data sets (the bar corresponding to <900nodes,100graphs> overdraws the scale; its evaluation time is 40 seconds)**

### 4.3.2. Loading performance

In this test we analyze loading performance. We measure the time it takes the tool to communicate with the metabase for loading each session and their components.

The test with the small data sets shows that loading time is linear in the number of graphs and slightly increases with the number of nodes. Figure 5.18 shows the test result. Times are bigger than evaluation ones; some data sets were loaded in almost a minute. Noise is also bigger, which is mainly due to Oracle internal routines (e.g. logging) and inter-process communication delays.

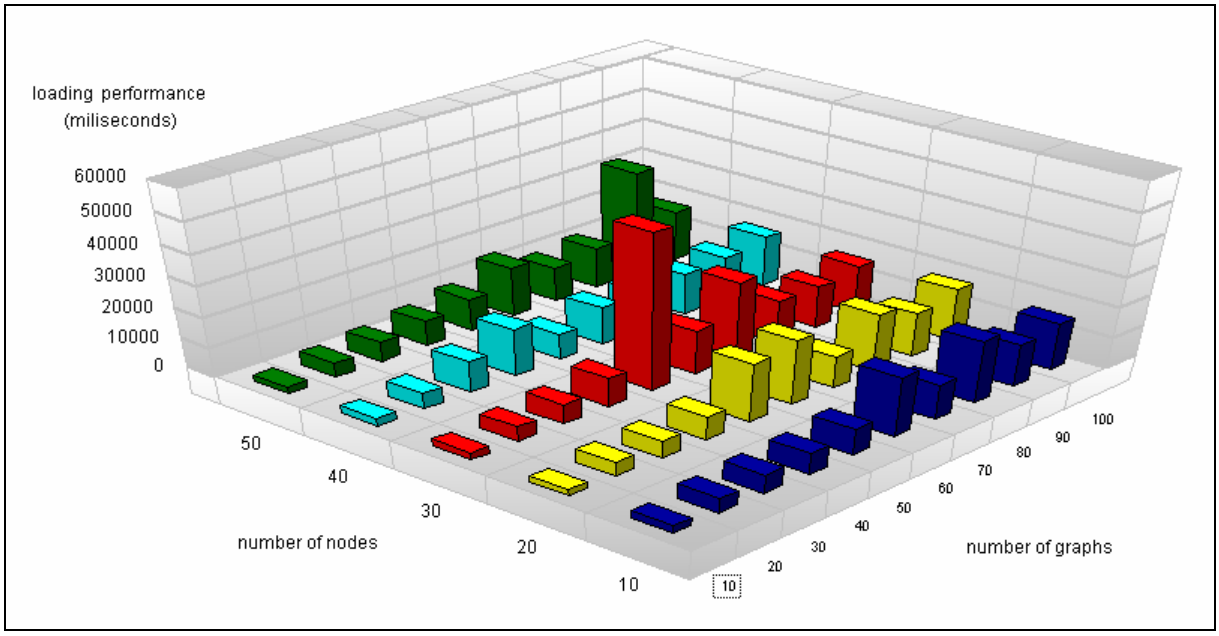


**Figure 5.18 – Loading times for the small data sets**

The test with the bulk data sets also shows that loading time is linear in the number of graphs. Figure 5.19 shows the test result. Loading can be performed in less than a minute for most data sets.

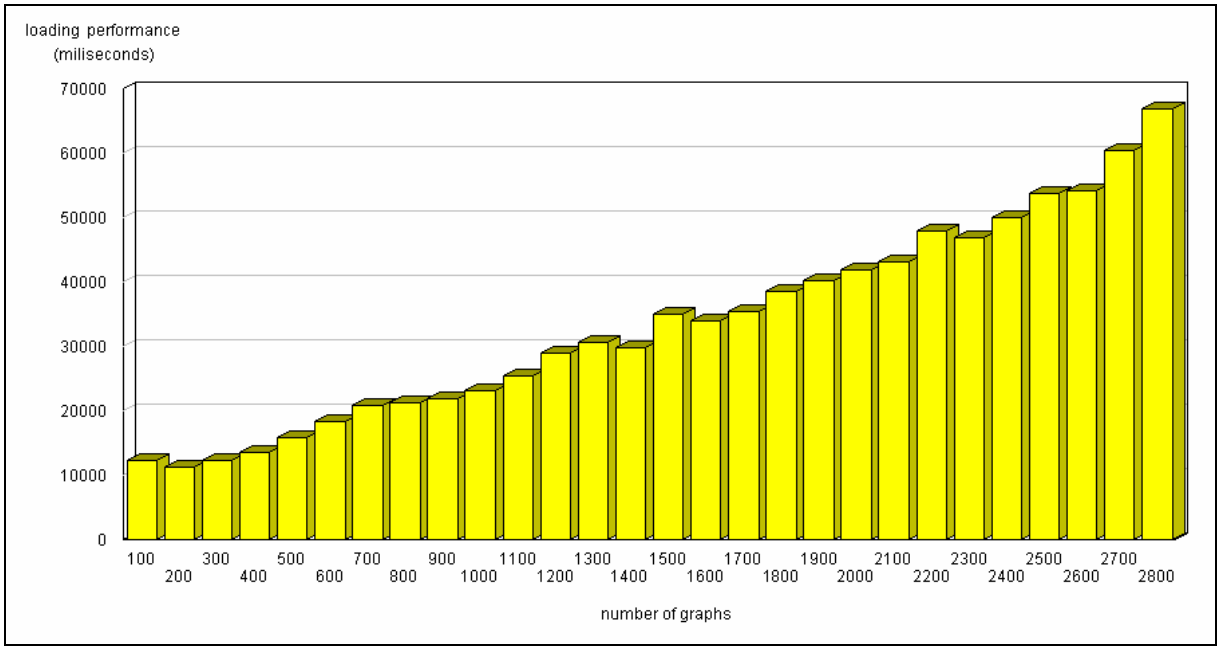


**Figure 5.19 – Loading times for the bulk data sets**

The test with the big data sets confirms that loading time is linear in the number of graphs and the number of nodes also for big graphs. However, the limit data sets (the biggest ones being supported) overdraw the linear scale. Figure 5.20 show the test result. The loading of some hundred graphs of some hundred nodes lasts some minutes.

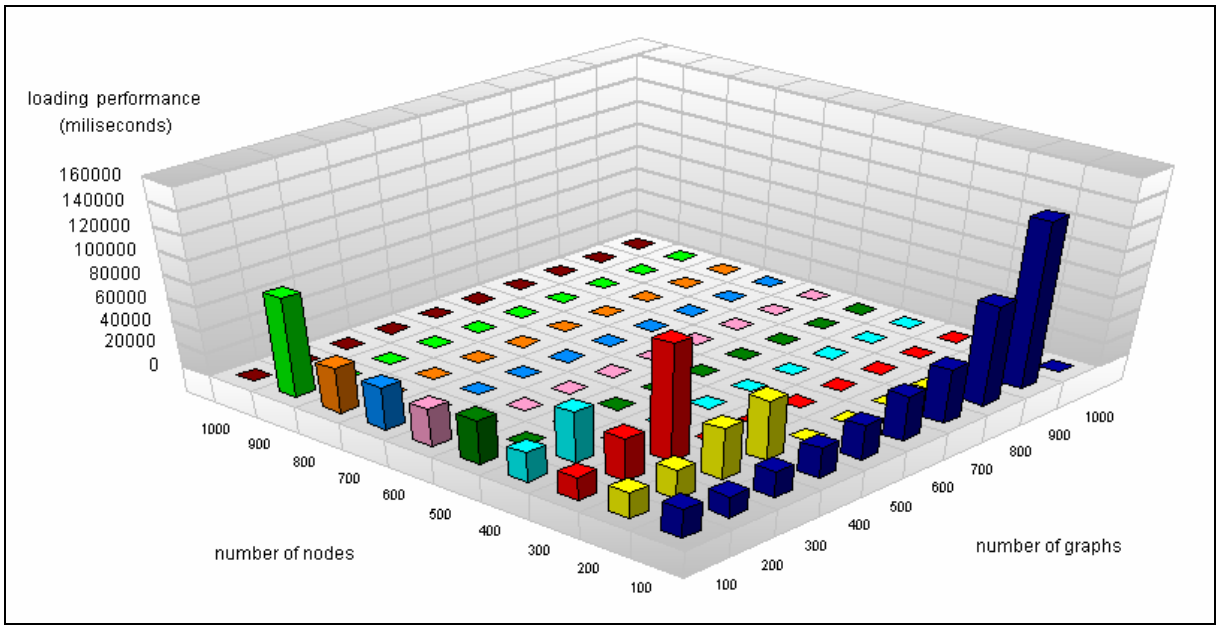


**Figure 5.20 – Loading times for the big data sets**

## 5. Conclusion

In this chapter we presented our experimentations with data freshness and data accuracy evaluation. We described the prototype of a data quality evaluation tool, DQE, which manages the proposed quality evaluation framework. The tools models the framework components, namely, data sources, data targets, quality graphs, properties, measures and quality evaluation algorithms.

We used the tool for evaluating data freshness and data accuracy in several application scenarios in order to validate our approach. Specifically, we described three applications: (i) an adaptive system for aiding in the generation of mediation queries, (ii) a web warehousing application retrieving movie information, and (iii) a data warehousing system managing information about students of a university. We showed how the DIS applications were modeled as quality graphs and how evaluation algorithms were instantiated to them. This experimentation allows validating the approach in real applications, specially, its ease of use for modeling DIS and properties and instantiating quality evaluation algorithms.

We also describe some tests for evaluating performance and limitations of the tool. We generated some data sets (quality graphs adorned with property values) and we executed a quality evaluation algorithm over each graph. The obtained results allow affirming that the tool can be used for large applications (modeling hundreds of graphs with hundreds of nodes each).