# Εξέλιξη Βάσεων Δεδομένων και Συντήρηση Εξαρτώμενων Εφαρμογών μέσω Επανεγγραφής Ερωτήσεων

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ
υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από τον

Πέτρο Μανούση

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Φεβρουάριος 2013

# Table of Contents

# List of Figures

# List of Tables

# List of Algorithms

# Abstract

Petros Manousis.

MSc, Computer Science Department, University of Ioannina, Greece.

Database Evolution and Maintenance of Dependent Applications via Query Rewriting.

Supervisor: Panos Vassiliadis

Data-intensive ecosystems are collections of databases and application programs that heavily depend on the underlying databases for their operation, thus, containing queries. The goal of this Thesis is to provide the means for the smooth evolution of the ecosystems in the presence of potential changes in their database part; specifically, we assess the impact of a potential change and present the result of the adaptation of the affected queries to its new structure. To this end, we trace all the components and interdependencies of the ecosystem via a single, uniform representation, which we call Architecture Graph. Our approach is also based on a language for annotating the ecosystem's modules with policies for their adaptation to future events. We provide a confluent algorithm for assessing the impact of a tested event as well as an algorithm for providing rewritings for views and queries of the ecosystem. All these algorithms have been incorporated into the existing Hecataeus system that allows the modeling, visualization, and evolution management of data-intensive ecosystems. In the context of this effort, the system was significantly extended with a new model for software modules (facilitating the increased modularity of the representation), along with novel, extensible parts responsible for policy annotation and impact assessment.

# Εκτεταμένη Περίληψη στα Ελληνικά

Πέτρος Μανούσης του Βασιλείου και της Αρετής.

MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων.

Εξέλιξη Βάσεων Δεδομένων και Συντήρηση Εξαρτώμενων Εφαρμογών μέσω Επανεγγραφής Ερωτήσεων.

Επιβλέπων καθηγητής: Παναγιώτης Βασιλειάδης

Τα οικοσυστήματα δεδομένων αποτελούνται από βάσεις δεδομένων και εφαρμογές οι οποίες εξυπηρετούν ερωτήσεις και συνεπώς εξαρτώνται σε μεγάλο βαθμό από αυτές. Στόχος αυτής της διατριβής είναι η παροχή αλγορίθμων για την ομαλή εξέλιξη των οικοσυστημάτων αυτών μετά από κάποια αλλαγή σε κάποιο τμήμα της βάσης δεδομένων. Ειδικότερα, αξιολογούμε την επίπτωση μιας επικείμενης αλλαγής και παρουσιάζουμε ως αποτέλεσμα τη νέα μορφή των επηρεαζόμενων (από την αλλαγή) ερωτήσεων της βάσης. Για το λόγο αυτό έχουμε αναγάγει όλα τα στοιχεία και τις εξαρτήσεις του οικοσυστήματος σε ένα ενιαίο γράφημα αναπαράστασης εκπροσώπησης το οποίο ονομάζουμε γράφημα αρχιτεκτονικής του οικοσυστήματος. Επιπλέον, η προσέγγισή μας βασίζεται σε μια γλώσσα που χρησιμοποιείται για την επισημείωση των δομικών στοιχείων (modules) με κανόνες. Παρέχουμε έναν αλγόριθμο για την εκτίμηση των επιπτώσεων που έχει μια πιθανή αλλαγή καθώς επίσης έναν αλγόριθμο για την επανεγγραφή των όψεων και ερωτημάτων του οικοσυστήματος. Οι αλγόριθμοι αυτοί έχουν ενσωματωθεί στο υπάρχον σύστημα Hecataeus το οποίο επιτρέπει την μοντελοποίηση, οπτικοποίηση και διαχείριση οικοσυστημάτων δεδομένων. Στο πλαίσιο αυτής της προσπάθειας, το σύστημα αυτό απέκτησε ένα νέο μοντέλο αναπαράστασης των δεδομένων καθώς επίσης και επεκτάσιμα κομμάτια κώδικα, υπεύθυνα για την επισημείωση των δομικών στοιχείων με κανόνες επί των αλλαγών, την εκτίμηση του εύρους των επιπτώσεων των αλλαγών και την επανεγγραφή του οικοσυστήματος ώστε να προσαρμοστεί στις αλλαγές αυτές.

# Chapter 1

# Introduction

Data-intensive ecosystems are configurations of database and software applications that interoperate. The main characteristic of a data-intensive ecosystem is the co-existence of (a) a central repository of data, typically in the form of a relational database including relations, views, constraints, as well as triggers and stored procedures, and (b) a set of software applications that require access to the central database, typically via queries to its views and relations. Naturally, ecosystems can comprise more than one data management repositories, not necessarily of relational nature (e.g., files, XML databases, sources of streaming data, etc); however, in the context of this work, we restrict to relational databases surrounded by applications that use queries over them.

To operate smoothly, an ecosystem must withstand change gracefully. Software maintenance amounts up to 60% of the resources spent on building an operating a software system [Pre00] and thus, it is of utmost importance for a system's life-cycle. In this context, the management of changes in a data-centric ecosystem is an important problem. In this thesis, we extend the state of the art concerning several research questions in the area of managing the evolution of data-intensive ecosystems.

*What does evolution of data-intensive ecosystems amounts at*? Evolution is of significance if it affects the syntactic correctness, the semantic validity, the operational effectiveness, or the administrative overhead of an ecosystem. Here are a few examples of possible changes:

- A certain attribute of the schema of a view is about to be deleted, as the administrator wants to simplify the definition of the view
- A new attribute is added to a relation, because new content is available
- The WHERE clause of a view is modified with an extra condition, to account for a new definition of the view's contents

- An index is dropped, as the administrator deems it useless
- A relation has grown too large and has to be moved to another table-space

Why are these changes of interest? First, some changes can lead to failure of existing applications. A deleted attribute might cause applications using it to crash. In this case, the applications' developers have to take care of the change: pinpoint its impact in their code, assess the necessity for the existence of this information in the applications and modify their applications accordingly. If things go wrong, this might even require negotiations with the DBA's to restore the deleted attribute. Second, applications can be affected semantically. If a new attribute is added to a relation it is possible that it contains important information that applications should be exploiting (and thus, have to be synchronized to the new contents of the relation). If the semantics of a view change, then the data delivered at the view's clients are different than the ones delivered before: in this case, the developers of the affected queries and applications would have to be notified and decide on whether the queries have to adapt to the new semantics of the view, or, they would have to retain the old semantics (again leading to the problem of compensating the change performed by the DBAs). Third, performance and administrations issues can affect the operation and administration of the ecosystem. An index is dropped and suddenly a large number of queries run unacceptably slow. A table is moved and suddenly the disk is full and other tables do not have enough space for their own insertions.

In all these occasions, we observe that a change performed by the DBA team can have several side-effects both for the team itself, the developers of applications of the ecosystem and the end-users. The problem in all the aforementioned events was that the change was performed before assessing its impact over the ecosystem. Therefore, addressing the impact assessment problem in advance of a potential change can be really valuable. *We would like to highlight that all our deliberations operate in a what-if analysis environment, where the DBA is committed to pre-assess the impact of his possible modifications, before actually performing them*; in other words, our environment is not tightly coupled to the databases, but rather a testing environment for potential, hypothetical modifications.

*How can we assess the impact of a change in a data intensive ecosystem*? In this Thesis, we build upon the state of the art to provide concrete results for the problem of impact assessment. We follow the model of architecture graphs [PVSV07, PVSV09, PVSV10] that capture all the inter-dependencies between the constructs of databases and the application queries via a graph. The graph models constraints, attributes, relations, views and queries along with their internal structure as the nodes of the graph. The edges of the graph denote dependency for data provision (e.g., between a view and a relation that populates it with data), part of relationships (e.g., between a relation and its attributes) and semantic relationships (e.g., the construction of the WHERE clause of a query as a tree of expressions). This way, the architecture graph

models all the components of a data-intensive ecosystem in a uniform way. One of the main utilities of the architecture graph is that facilitates impact assessment for potential changes in the ecosystem: whenever a potential change is tested over the architecture graph, the graph allows us to predict the impact by recursively following edges between affected nodes.

*Is it possible to regulate change in a data-intensive ecosystem*? Are we helpless in managing potential changes in the core of the ecosystem? If an application developer is really adamant on retaining the structure and semantics of a database view, it is possible that he states this requirement somehow in the architecture graph, to prevent possible modifications? As previous research [PVS+08, PVSV09] has demonstrated, it is possible to define policies for handling events. We annotate the nodes of the architecture graph with policies for handling events. For example, we can annotate the relations of the graph with policies for handling the potential deletion of an attribute. We assign one of two possible reactions to this event, and specifically, *block* to veto the event and demand that the relation retains its previous structure or *propagate* to allow the event and make the relation ignite the recursive notification of all the affected software modules in the graph.

This thesis extends the results of the state of the art in modeling data-intensive ecosystems and assessing the impact of changes in several ways:

- First, it extends the modeling of ecosystem *modules* (relations, views, queries) by encapsulating their structure within input and output schemata. Previous research [PVSV09] dealt with the architecture graph as a single unified graph where software modules did not have boundaries; for example, a query that used the attributes of a relation would directly access these attributes via data-dependency edges. Although this produces a smaller graph, unfortunately is hinders the separation of modules. As we shall see, this separation will allow us to provide theoretical guarantees for our algorithms. At the same time, it inspires the principles of modular design into the modeling of architecture graphs.

- Second, this thesis describes the concrete implementation of the algorithms proposed in [PVS11] for impact assessment. The modular design guarantees that the propagation of a potential evolutionary event over the graph (a) terminates and (b) is performed in a way that is independent of the internal processing of events within modules.

- Third, this thesis has implemented a concise language for policy annotation of the ecosystem modules. We follow [PVS+08] in spirit and provide a rule-based language that (a) allows the annotation of all the nodes of the graph with default policies, (b) completely covers all the space of events (i.e., all nodes have a policy to handle *any* possible event that comes to them), and, (c) allows the users to customize reaction with policies other than the default ones for individual modules. We handle events and policies that alter the

structure and semantics of the ecosystems by modifying the schema of the database and the semantics of views (performance related events are outside the scope of this work).

*Once the impact of a change has been assessed, is it possible to see how the ecosystem will look like if the change is eventually performed*? Even with the presence of policies, it is possible that a potential modification in the database affects several queries and views that are willing to accept it and adapt to the new structure or semantics of the database. Then, the question that has to be answered is "what will the new structure and semantics of all the affected modules look like?". *The core result of this Thesis is the provision of algorithms that perform the rewriting of affected modules to adapt to the potential event*. Specifically, our method works in the following three steps:

1. Impact assessment. Given a potential event, a status determination algorithm makes sure that the nodes of the ecosystem are assigned a status concerning (a) whether they are affected by the event or not and (b) what their reaction to the event is (block or propagate).

2. Calculation of variants. Assume a view used by two queries is altered. Assume also that the first query vetoes the change and requires the structure and semantics of the old view to remain, whereas the second concedes to the change and states it will adapt to the new structure and semantics of the view. *The co-existence of blocker and adapter data consumers of an affected module signifies the need to retain both the old and the new version of the module, whenever this is possible*. To this end, we introduce an algorithm that checks the affected parts of the graph in order to highlight affected nodes with whether they will adapt to a new version or retain both their old and new variants.

3. Module Rewriting. Once the status and number of variants has been determined for the modules of the graph, we need to implement the rewritings. This is heavily dependent upon the nature of the event (obviously, a query adapts differently to the removal of an attribute and differently to the addition of an attribute, let alone changes in semantics). Our algorithm visits affected modules sequentially and performs the appropriate restructurings of nodes and edges.

The introduction of algorithms for module rewriting in the presence of policies is not the only contribution of this Thesis in this area. We can also list the following extra contributions in terms of principles and software construction:

- *The principle of locality independence*. Both impact assessment and rewritings take place within modules, without referring to the entire graph for auxiliary help. The idea is that each module waits until all the notifications that affect it are produced and then, collects them and performs status determination, variant calculation and rewriting in isolation

of the rest of the graph. We can theoretically guarantee that it is possible to prioritize the enactment of each module in the appropriate order, such that all its notifications are present. We can also guarantee that impact assessment is the same independently of the order of notification generation. The principle of locality independence states that, once orchestrated and scheduled for activation, we can treat each module on its own. This lays the groundwork for future explorations concerning the possibility for parallel processing of non-dependent modules and management of transactions of changes applied over the graph.

- *The principled construction of software in order to facilitate the locality independence*. Constructing the software that performs the rewritings in a way that each module "wakes" an impact assessment/rewriting module dedicated to itself is a challenging task in terms of software engineering. We imposed the extra requirement of extensibility with respect to the types of events handled. The constructed software parts introduce the notion of *maestro*, which is a module that is (a) responsible for a certain task (impact assessment or rewriting), (b) over a specific type of nodes (relations, views, queries), and (c) with an extensible palette of events it can handle.

All the algorithms have been implemented and integrated within an existing software tool, Hecataeus[1] [PAVV08, PVSV10]. Hecataeus can be used for the construction and visualization of the architecture graph, its annotation with policies, and the testing of hypothetical events over the annotated graph. The last task includes both impact assessment and rewriting. Moreover, Hecataeus is equipped with metrics management that allows the assessment of graph-theoretic metrics in order to identify regions of the graph that might be sensible to evolution events.

We have experimented with our methods in different ecosystems, policy assignments and event workloads and assessed their effectiveness and efficiency. A first observation on our experiments, confirms the benefits introduced by our method concerning the effort performed by the application developers and administrators of the ecosystem. In the absence of our system, the typical developer would have to perform at least 25% of routine, useless checks to views and queries that are not related to the event at all; on average, the number of useless checks rises in the area of 90%.

A second observation has to do with the amount of rewriting: in all occasions, there have been several modules that had to be rewritten. Although the numbers are not particularly high, ranging from 1 to 4.5 modules, the automation of the work, equips the involved stakeholders with correctness guarantees that would otherwise be non-existent.

In terms of time, all the experiments show a completion of the tested changes as fractions of a second; specifically, the average times range in the area of 0.7 millisecond, whereas the

[1] http://www.cs.uoi.gr/ pvassil/projects/hecataeus/

maximum times do not go beyond 5 millisecond. Although the time needed to perform impact assessment and rewriting is not significant, if we inspect the way different modules respond to different events, it is clear that the time taken to perform an event can vary a lot as a result of the popularity of a module as data-provider with its policy on the event. As expected, excessive peaks in impact assessment and rewriting time concern modules with high fan-in of dependent modules; these are clearly candidates where evolution should rather be blocked.

Apart from the above contributions, we can also mention that in the context of this Thesis, the internals of Hecataeus were re-engineered to adapt to the new model and the user-interaction GUI parts were reshaped to allow the efficient definition of policies and the management of events. The tool was also enriched with the notion of "project" that introduced structured storage of the graphs contents, policies and visualization layout for future reuse.

**Roadmap.** The structure of this Thesis is as follows. In Chapter 2, we give the background modeling for the architecture graph, policies and events. In Chapter 3, we discuss impact assessment, and, in Chapter 4, we discuss the module rewriting. In Chapter 5, we discuss the software architecture of our contribution, and, in Chapter 6, we present the experiments that we have performed in order to study the behavior of the introduced algorithms and their benefits. In Chapter 7, we present related work. We conclude in Chapter 8, along with insights for future work.

# Chapter 2

# Background: a formal model for architecture graphs, evolutionary events and policy annotation

---

To assess the impact of a potential change over the data centric ecosystem, we construct a graph of modules (relations, queries and views) where data consuming nodes are linked with edges to their providers. Whenever an event is applied over a module, the module has to assess the impact of the event and notify its consumes. This recursive process allows us to assess the impact of the event over the entire ecosystem. Naturally, to facilitate this process, we need to establish a formal model for the main constituents of the problem and its solution. So, before proceeding with the algorithmic parts of the adaptation process, in this Section, we present the formal background for the modeling of architecture graphs, along with the space of possible events and policy annotations. First, we present how relations, views and queries construct the architecture graph of the ecosystem. Then, we move on to present the space of possible events that can be applied to the nodes of the graph, either directly by the user (initiating the entire process of assessing the impact of an event) or transitively, as modules affected by the event notify other modules that depend on them for the change. Moreover, in order to regulate the propagation of events over the graph, we present the language for policy annotations, along with its semantics and the rules for policy overriding.

Before proceeding it is worth noting that our modeling improves on the previous versions of Hecataeus ([PVSV10, PVSV09]) by enforcing that *all* modules have a well defined scope, "fenced" by input and output schemata. Previous models allow queries and views to directly refer to the nodes representing the attributes of the involved relations. In our model, all software modules employ an output schema and in the case of queries and views a set of input schemata.

This way, the internals of software modules can be isolated from the rest of the graph. On the practical side, this modeling also facilitates the correct passing of notifications from one module to another.

## 2.1 Architecture graph

Our modeling technique, following [PVS11], represents all the aforementioned database constructs as a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, which we call architecture graph of the ecosystem. Next, we briefly present the components of the architecture graph. We start with the high level constructs, such as relations and queries, which we call modules of the architecture graph, and then we move on to discuss their main properties.

**Modules.** A module is a semantically high level construct of the ecosystem; specifically, the modules of the ecosystem are relations, views and queries. These modules are disjoint and they are connected through edges concerning provider or semantic-level relationships, as we shall see in the sequel.

Every module defines a scope: within the scope of a module a sub-graph of the architecture graph is assumed. For example, the attributes and local constraints of a relation live within the relation's scope. A scope is nothing more than a set of part-of relationships that connect the component (which is expressed as a node) with its constituents.

**Relations, R.** Each relation $R$ ($\Omega_1$, $\Omega_2$,..., $\Omega_n$) in the database schema is represented as a directed graph, which comprises: a node, $R$, representing the relation; *an output schema node*, $R\_SCHEMA$, representing the relation's output schema; n attribute nodes $\Omega_{i=1...n}$, one for each of the attributes and n+1 schema relationships $E_{i=1...(n+1)}$, directing from the schema node towards the attribute nodes, indicating that the attribute belongs to the relation's output schema and one directing from the relation node towards the output schema node indicating that the output schema belongs to the relation.

**Conditions, C.** Conditions refer both to selection conditions of queries and views and constraints of the database schema. We consider three classes of atomic conditions that are composed through the appropriate usage of an operator *op* belonging to the set of classic binary operators, *op* (e.g., $<$, $>$, $=$, $\leq$, $\geq$, $!=$, $IN$, $EXISTS$, $ANY$): $\Omega$ *op* constant; $\Omega$ *op* $\Omega$' and $\Omega$ *op* $Q$ where $\Omega$, $\Omega$' are attributes of the underlying relations and $Q$ is a query.

A condition node is used for the representation of the condition. Graphically, the node is tagged with the respective operator and it is connected to the operand nodes of the conjunct clause through the respective operand relationships, $O$. Composite conditions are easily constructed by tagging the condition node with a Boolean operator (e.g., AND or OR) and the respective edges to the conditions composing the composite condition.

Well-known constraints of database relations – i.e., primary/foreign key, unique, not null, and check constraints – are easily captured by this modeling technique. Foreign keys are subset relations of the source and the target attribute, check constraints are simple value-based conditions. Primary keys, which are unique-value constraints, are explicitly represented through a dedicated node tagged by their names and a single operand node.

**Queries, Q.** The graph representation of a Select - Project - Join - Group By (SPJG) query involves:

(a) a new node representing the query, named *query node*,

(b) a set of *input schemata nodes* (one for every table appearing in the FROM clause). Each input schema comprise the set of attributes that participate in the syntax of the query (i.e., SELECT, WHERE clause, etc.)

(c) an *output schema node* comprising the set of attributes present in the SELECT clause.

(d) a *semantics* node as the root node for the sub-graph corresponding to the semantics of the query (specifically, the WHERE and GROUP-BY part), and,

(e) *attribute nodes* belonging to the various input schemata and output schemata of the query.

The query graph is therefore a directed graph connecting the query node with the high level schemata and semantics nodes. The schema nodes are connected to their attributes via schema relationships. In order to represent the relationship between the query graph and the underlying relations, we resolve the query into its essential parts: SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY. In our current model, we support the representation of the first four parts via a dedicated sub-graph (HAVING and ORDER BY are parts of future work).

*Select part*. Each query is assumed to own an output schema that comprises the attributes, either with their original or with alias names, appearing in the SELECT clause. In this context, the SELECT part of the query maps the respective attributes of the input schemata to the attributes of the query's output schema through <map-select> edges, directing from the output attributes towards the input schema attributes. We denote this subset of the edges of the graph by $\mathbf{E_M}$.

*From part*. The FROM clause of a query can be regarded as the relationship between the query and the relations (or views) involved in this query. Thus, the relations included in the FROM part are combined with the input schemata of the query node through from edges, directing from the nodes of the appropriate input schemata towards the output schema nodes of the relation/view nodes. We denote this subset of the edges of the graph by $\mathbf{E_F}$. The input schemata of the query comprise only the attributes of the respective relations that participate

in any way in the query; the attributes of the input schemata are connected to the respective attributes of the provider relations or views via map-select relationships.

*Where part.* We assume that the WHERE clause of a query is in conjunctive normal form. Thus, we introduce directed edge, namely where relationships, $\mathbf{E_W}$, starting from the semantics node of a query towards an operator node corresponding to the conjunction of the highest level. Then, there is a tree of nodes hanging from this conjunction as previously described for composite constraints. The edges are operand relationships as mentioned above among binary comparators, boolean operators, input attributes and constants.

*Group and Order By part.* For the representation of aggregate queries, we employ two special purpose nodes: (a) a new node denoted as GB, to capture the set of attributes acting as the aggregators and (b) one node per aggregate function labeled with the name of the employed aggregate function; e.g., COUNT, SUM, MIN.

For the aggregators, we use edges directing from the semantics node towards the GB node that are labeled <group-by>, indicating group-by relationships, $\mathbf{E_G}$. The GB node comprises separate children nodes for all attributes acting as aggregators. These edges are schema relationships, which are additionally tagged according to the order of the aggregators; we use an identifier $i$ to represent the i-th aggregator. Each of these attribute nodes is connected with the respective input attributes with a <map-select> edge. Moreover, for every aggregated attribute in the query's output schema, there exists an edge directing from this attribute towards the aggregate function node as well as an edge from the function node towards the respective input attribute. Both edges are labeled <map-select> and belong to $\mathbf{E_M}$, as these relationships indicate the mapping of the query attribute to the corresponding relation attribute through the aggregate function node.

**Functions, F.** Functions used in queries are integrated in our model through a special purpose node $F$, denoted with the name of the function. Each function has an input parameter list comprising attributes, constants, expressions, and nested functions, and one (or more) output parameter(s). The function node is connected with each input parameter graph construct, nodes for attributes and constants or sub-graph for expressions and nested functions, through an operand relationship directing from the function node towards the parameter graph construct. This edge is additionally tagged with an appropriate identifier $i$ that represents the position of the parameter in the input parameter list. An output parameter node is connected with the function node through a directed edge from the output parameter towards the function node.

**Views, V.** Views are treated as queries; however the output schema of a view can be used as input by a subsequent view or query module.

**Summary.** A summary of the architecture graph is a zoomed-out variant of the graph that comprises only of modules and their schema nodes as nodes and edges denoting relationships between modules and their schemata and any possible form of provider relationship between

modules. Formally, a summary is a directed acyclic graph $\mathbf{G_s} = (\mathbf{V_s}, \mathbf{E_s})$, with $\mathbf{V_s}$ comprising the graph's module nodes including their schema nodes and $\mathbf{E_s}$ comprising relationships between modules and their schemata and pairs of providers and consumers as from-relationship edges, $\mathbf{E_F}$.

**Example.** The following example shows a small university database. The database contains information on semesters, standard, recurring data for the courses offered by a department, specific data for the courses offered by the department in a particular semester, as well as information for students and their transcript – i.e., what course they have enrolled to and with what grade. The names of the relations and their attributes are self-explanatory.

```
CREATE TABLE semester
(
    mid INTEGER,
    mdescr VARCHAR(50),
    PRIMARY KEY (mid)
);

CREATE TABLE coursestd
(
    csid INTEGER,
    csname VARCHAR(50),
    cspts FLOAT,
    PRIMARY KEY (csid)
);

CREATE TABLE course
(
    cid INTEGER,
    csid INTEGER,
    mid INTEGER,
    FOREIGN KEY (mid) REFERENCES semester (mid),
    FOREIGN KEY (csid) REFERENCES coursestd (csid),
    PRIMARY KEY (cid)
);

CREATE TABLE student
(
```

```
    sid INTEGER,
    sname VARCHAR(50),
    PRIMARY KEY (sid)
);

CREATE TABLE transcript
(
    cid INTEGER,
    sid INTEGER,
    tgrade FLOAT,
    FOREIGN KEY (sid) REFERENCES student (sid),
    FOREIGN KEY (cid) REFERENCES course (cid),
    PRIMARY KEY (cid, sid)
);
```

On top of this set of relations, we define two views and two queries. The first view, V_COURSE, combines three relations: SEMESTER, COURSESTD and COURSE into a single view that contains both the identifiers and the descriptions of the involved entities. The second view: V_TR joins V_COURSE with the relation TRANSCRIPT, resulting in a view that outputs all the information needed for every student's enrollment. Then, we have two queries. The first query performs a self-join over view V_TR and presents a report that compares the grades for two courses, DB_I and DB_II for those students who enrolled in both of them. The second query reports the average grade (i.e., over successfully passed courses) for every student; the report requires students names, so the relation STUDENT is joined to the view V_TR.

Views:

```
CREATE VIEW v_course AS
SELECT semester.mid, semester.mdescr, coursestd.csid, coursestd.csname,
course.cid
FROM semester, coursestd, course
WHERE semester.mid=course.mid AND coursestd.csid=course.csid;

CREATE VIEW v_tr AS
SELECT v_course.mid, v_course.mdescr, v_course.csid, v_course.csname,
v_course.cid, transcript.sid, transcript.tgrade
FROM v_course, transcript
```

```
WHERE v_course.cid=transcript.cid;
```

Queries:

```
SELECT v1.sid, v1.csname AS csname1, v1.tgrade AS tgrade1, v2.csname AS
csname2, v2.tgrade AS tgrade2
FROM v_tr AS v1, v_tr AS v2
WHERE v1.mid=v2.mid AND v1.sid=v2.sid AND v1.csname='B_I' AND v2.csname=
'DB_II';
```

```
SELECT student.sid, student.sname, avg(v_tr.tgrade) AS gpa
FROM v_tr, student
WHERE v_tr.sid=student.sid AND v_tr.tgrade > 4 / 10
GROUP BY student.sid, student.sname;
```

Figure 2.1 graphically depicts the graph structure of the relation COURSESTD. Observe that the module of the relation comprises the following kinds of nodes: (a) a node for the relation per se, (b) a node for its schema (actually, we treat the schema of a relation as its *output* schema), (c) three nodes for its attributes. The edges of the sub-graph include directed *part-of edges* from container, higher-level nodes to their lower-level constituents, and specifically, (a) the relation with its output schema and (b) the output schema with the attributes.

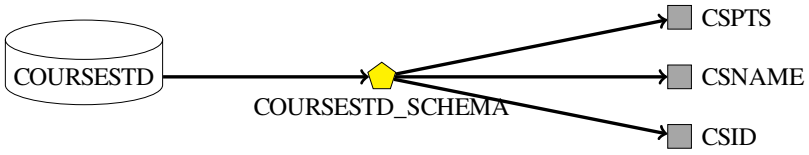



**Figure 2.1:** Graph representation of COURSESTD relation

Figure 2.2 graphically depicts the graph structure of a view. The view has a composite structure of four parts:

1. The view module comprises two *Input* schemata, named after their providers (in this case, the relation TRANSCRIPT and the view V_COURSE. Each input schema comprises all the attributes that are necessary for the other parts of the view definition to operate. In other words, the input schemata contain only the attributes that are used by selection conditions or groupers in the *Semantics* schema, or as output attributes under the *Output* schema of the view; the rest of the attributes of the module's providers are not included in the module's input schemata. The part-of edges are directed again from container to constituent nodes

2. The *Semantics* schema contains a single atomic equality operator representing the join condition of the SQL query defining the view. Observe the edges linking the equality operator to the operands of the equality and the edge linking the semantics schema to the equality operator.

3. The *Output* schema comprises the attributes appearing in the SELECT clause of the query. Observe the provider edges stemming from the attribute nodes of the output schema and reaching their respective provider nodes in the input schemata of the module. *In our modeling, data provision is modeled as a dependency edge; thus data consumers point towards their providers*.

4. Finally, a node representing the view per se at the highest level of abstraction.

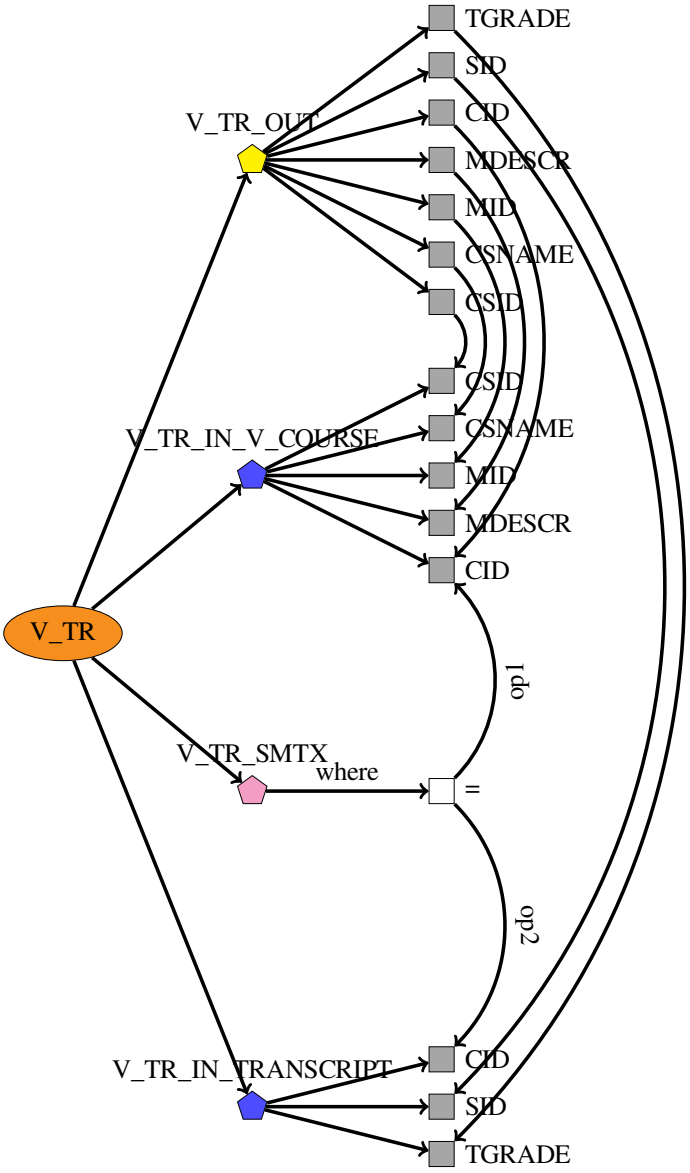



**Figure 2.2:** Graph representation of V_TR view

Queries are represented in a similar fashion with views. Figure 2.3 graphically depicts the graph structure of a query. The query has a composite structure of four parts:

1. The query module comprises two *Input* schemata, named after their providers (in this case, the relation STUDENT and the view V_TR. In a similar vein to the input schemata of the view of the previous example, each input schema comprises all the attributes that are necessary for the other parts of the query definition to operate (in figure 2.2 we see that V_TR view has 7 attributes in its output schema, while we have in our input schema

only 2 of them).

2. The *Semantics* schema contains an equality operator representing the join condition of the SQL query defining and an atomic grater than operator wanting an attribute to have values grater than the result of a mathematical operation. Observe the edges linking the equality operator to the operands of the equality and the edge linking the semantics schema to the equality operator. The *Semantics* schema also contains the *group by* node. Observe how the annotation of the edges streaming from the group by node denotes the order of the groupers. In our example we group by student.sid first and student.sname second. Finally, the *Semantics* schema contains the aggregate functions that are used in the query. In our example we use the aggregate function avg on attribute v_tr.tgrade. Observe how the semantics schema connects to the node representing the aggregate function; this node, in turn, depends on some input attribute, linked via the appropriate edge and populates an attribute node at the output schema, again via an appropriate edge.

3. The *Output* schema comprises the attributes appearing in the SELECT clause of the query. Observe the provider edges stemming from the attribute nodes of the output schema and reaching their respective provider nodes in the input schemata of the module or the aggregate functions of the semantic schema.

4. Finally, a node representing the query per se at the highest level of abstraction.

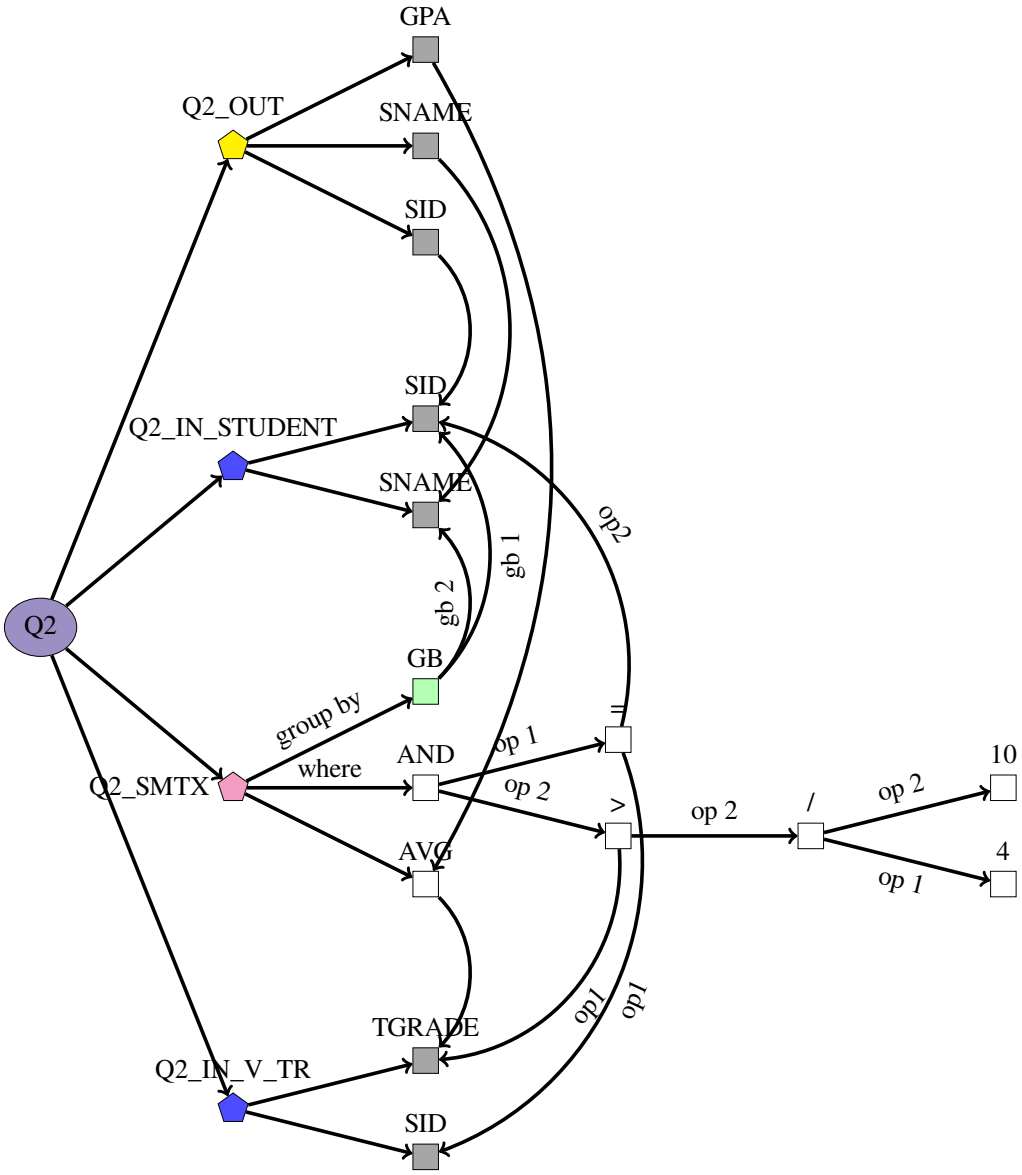



**Figure 2.3:** Graph representation of Q2 query

Figure 2.4 represents the small university database example displaying all relations, views and queries at high-level module mode. From the definition of V_COURSE view we see that COURSE, COURSESTD and SEMESTER are used in the FROM clause denoting that V_COURSE depends on these three relations. V_TR view uses V_COURSE and TRANSCRIPT relation in its FROM clause denoting that the view depends on these two modules. The first query uses V_TR view in its FROM clause (two times) denoting that the query depends on that view. Finally, the second query uses V_TR view and STUDENT relation

in its FROM clause denoting that the query depends on the view and the relation respectively.

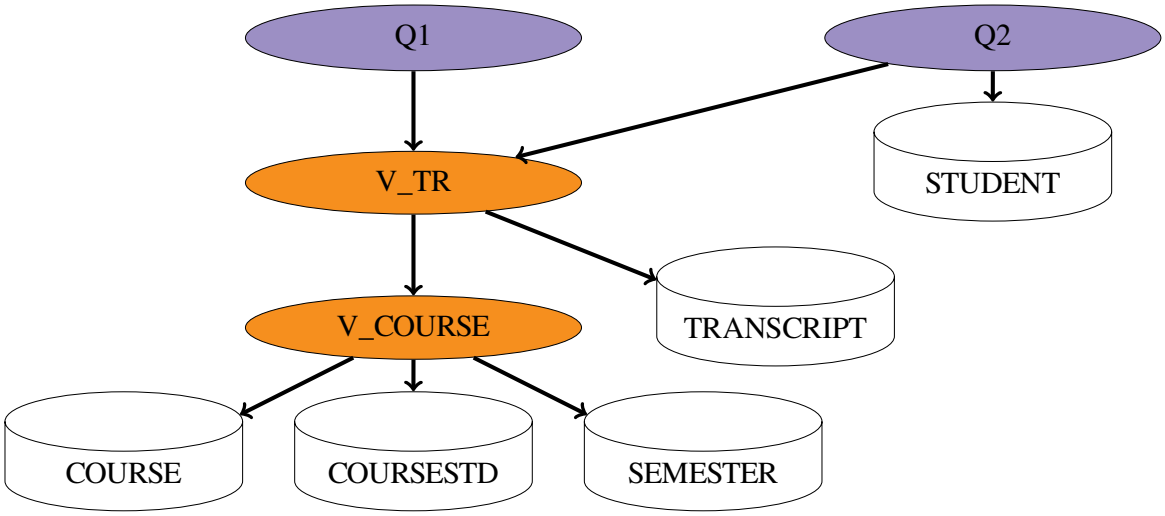



**Figure 2.4:** Reference example for a university database and its ecosystem

## 2.2 Events

In this section we list the set of possible events that our method handles. We organize our discussion by classifying these events in three classes: (a) events pertaining to relations, (b) events pertaining to views or queries, and (c) events that occur as one module notifies another for the event it just received. In our discussion, for reasons of clarity and intuition, we relate each event to its respective SQL command, whenever possible.

We can classify the impact of an event as *structural* whenever the exported schemata and their attributes are changed in terms of structure or naming. At the same time, the impact of an event is *semantic* whenever the internals of the semantics schema (i.e., the WHERE or the GROUP-BY clause of the respective SQL query) change.

### 2.2.1 The space of events that pertain to relations

The events that pertain to relations are the following:

**ADD_ATTRIBUTE:** in this case, a Relation should obtain another column (ALTER TABLE <TABLE> ADD <COLUMN> <TYPE>).

**DELETE_ATTRIBUTE:** in this case, a Relation should drop a column (ALTER TABLE <TABLE> DROP COLUMN <COLUMN>). This event is system generated and only

occurs in the initialization of messages that we discuss in subsection 3.1.3. This initialization is currently a function at evolution graph class.

**RENAME_ATTRIBUTE:** in this case, a Relation should rename a column (ALTER TABLE <TABLE> RENAME COLUMN <OLD> TO <NEW>). This event is system generated and only occurs in the initialization of messages.

**DELETE_SELF:** in this case, a Relation will be deleted (DROP TABLE <TABLE>).

**RENAME_SELF:** in this case, a Relation will be called with a new name from now on (ALTER TABLE <TABLE> RENAME TO <NEW>).

### 2.2.2 The space of events that pertain to views and queries

The events that pertain to Views/Queries are the following:

**ADD_ATTRIBUTE:** in this case, a Query/View should have another attribute (column, aggregate function or value) in its output (SELECT X, Y instead of SELECT X).

**DELETE_ATTRIBUTE:** in this case, a Query/View should have less attributes in its output (SELECT X instead of SELECT X, Y). This event is system generated and only occurs in the initialization of messages.

**RENAME_ATTRIBUTE:** in this case, an attribute is going to be called with a new name from now on (SELECT X AS Y instead of SELECT X). This event is system generated and only occurs in the initialization of messages.

**DELETE_SELF:** in this case, a View will be deleted (DROP VIEW <VIEW>).

**RENAME_SELF:** in this case, a View will be called with a new name from now on (ALTER VIEW <OLD> RENAME TO <NEW>).

**ALTER_SEMANTICS:** in this case, a View is going to have another WHERE clause or another GROUP BY clause (REPLACE VIEW <VIEW> AS SELECT <COLUMN> FROM <TABLE> WHERE <COLUMN>=<VALUE> instead of SELECT <COLUMN> FROM <TABLE>).

### 2.2.3 The space of events that occur due to the propagation of the impact of a potential change over the graph

Besides those events, there is the following list of events that accrue from the flow of an event to the graph:

**ADD_ATTRIBUTE_PROVIDER:** this event is generated by a module in order to inform its consumers that the module has added an attribute to its output schema.

**DELETE_PROVIDER:** this event is generated by a module in order to inform its consumers that this module has deleted one or all its attributes (we will see more at messages in section 3.1.3) from its output schema.

**RENAME_PROVIDER:** this event is generated by a module to inform its consumers that the module itself or one of the attributes that exist in output schema of the module want to change their name.

**ALTER_SEMANTICS:** this event is generated by a module to inform its consumers that the semantics (as described previously: change of WHERE or/and GROUP BY clause) of a module have changed.

### 2.2.4 A summary of the space of events

Summarizing the previous subsections, in that Table 2.1 we present how each category of nodes can sustain its own set of incoming evolution events either due to the user's input or due to the system's event propagation mechanism. Note that the each type of nodes is linked to kinds of events it can *receive*; as we shall see later, event emission is performed only by modules (once a module has internally processed all its arriving events).

| | R.OUT | R.OUT.ATTRS | QV.IN | QV.IN.ATTRS | QV.OUT | QV.OUT.ATTRS | QV.SMTX |
|---|---|---|---|---|---|---|---|
| DELETE_SELF | ✓ | ✓ | | | ✓ | ✓ | |
| RENAME_SELF | ✓ | ✓ | | | ✓ | ✓ | |
| ADD_ATTRIBUTE | ✓ | | | | ✓ | | |
| DELETE_PROVIDER | | | ✓ | ✓ | | | |
| RENAME_PROVIDER | | | ✓ | ✓ | | | |
| ADD_ATTRIBUTE_PROVIDER | | | ✓ | | | | |
| ALTER_SEMANTICS | | | ✓ | | | | ✓ |

**Table 2.1:** Incoming events that each node type can receive.

Notation: In Table 2.1, R stands for relations, QV stands for queries and views and ATTRS stands for attributes (of any kind); OUT/SMTX/IN stand for output/semantics/input schemata, respectively.

## 2.3 Policies

In this section we present the management of policies for regulating the response of the ecosystem to a hypothetical event. First, we discuss which are the supposed policies and their se-

mantics. Second we present a language for the declarations of these policies in an ecosystem. Moreover we introduce a *complete* set of policy declarations that covers *all* possible events in an ecosystem. Finally we return to our reference example for a university database and its ecosystem to discuss policy annotation of the architecture graph.

**Policies for each kind of node**. Whenever a node receives an event that concerns either itself or its constituents (e.g., the attributes of a schema), the node has to respond with a reaction to the incoming event. The policy of a node for responding to an incoming event can be one of the following:

- PROPAGATE, which means that the node accepts the change and will adapt to the new reconfiguration of the graph that will be produced by the propagation of the event over the graph, or,
- BLOCK, which means that the node wants to retain the previous structure and semantics of the graph

**Requirements for the policy annotation language.** We wish to provide a language that facilitates the annotation of the ecosystem with policies in a way that increases the users' productivity. To this end, we employ a rule–based language, in the spirit of [PVS+08]. Moreover, we address the following usability requirements:

- *Completeness*: we need to be able to define annotations for all the possible events that can arrive to a node
- *Conciseness*: we need to make it easy for the user to annotate the ecosystem with policies, without going to great lengths of coding
- *Customizability*: we need to allow the users to define rules that are tailored to specific subparts of the ecosystem.

**Language for policy annotation**. In order to annotate the graph, we utilize an annotation language that follows [PVS+08] in its philosophy. The space that the language has to cover is defined by the available combinations of events and node types; thus, each type of nodes can (in fact, has to) be annotated by a rule for its response to an incoming event. The language simply comprises rules that abide to the following structure:

<receiver node> : `on` <event> `then` <policy>

where:

1. <receiver node> can be any of the following:
   (a) QUERY.OUT.SELF standing for the node representing the output schema of all queries
   (b) QUERY.OUT.ATTRIBUTES standing for the nodes representing the attributes of the output schema of all queries
   (c) QUERY.SMTX.SELF standing for the node representing the semantics tree of all queries
   (d) QUERY.IN.SELF standing for the node representing the input schema(-ta) of all queries
   (e) QUERY.IN.ATTRIBUTES standing for the nodes representing the attributes of the input schema(-ta) of all queries
   (f) VIEW.OUT.SELF standing for the node representing the output schema of a view
   (g) VIEW.OUT.ATTRIBUTES standing for the nodes representing the attributes of the output schema of all views
   (h) VIEW.SMTX.SELF standing for the node representing the semantics tree of a view
   (i) VIEW.IN.SELF standing for the node representing the input schema(-ta) of a view
   (j) VIEW.IN.ATTRIBUTES standing for the nodes representing the attributes of the input schema(-ta) of all views
   (k) RELATION.OUT.SELF standing for the node representing the output schema of a relation
   (l) RELATION.OUT.ATTRIBUTES standing for the nodes representing the attributes of the output schema of all relations
   (m) <NAMED SCHEMA NODE>.ATTRIBUTES standing for the nodes representing the attributes of the <named schema node> of the graph.
   (n) <NAMED NODE> standing for the <named node> node of the graph.
2. <event> can be any of the events listed in section 2.2, with the constraint it has to be applicable to its accompanying <node type> (as dictated by Table 2.1)
3. <policy> can be either PROPAGATE or BLOCK

**Conciseness and completeness considerations**. We achieve the conciseness requirement by providing a small list of generic rules that cover all possible combinations of events and nodes. To address the completeness requirement, we have to be certain that we cover the entire

space of the combination of events with their receiving nodes. To this end, we provide a complete set of rules that once completed by the user, guarantee that all nodes are able to address any possible event that arrives to them.

QUERY.OUT.SELF: on ADD_ATTRIBUTE then <policy>;
QUERY.OUT.SELF: on DELETE_SELF then <policy>;
QUERY.OUT.SELF: on RENAME_SELF then <policy>;
QUERY.OUT.ATTRIBUTES: on DELETE_SELF then <policy>;
QUERY.OUT.ATTRIBUTES: on RENAME_SELF then <policy>;
QUERY.OUT.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
QUERY.OUT.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
QUERY.IN.SELF: on DELETE_PROVIDER then <policy>;
QUERY.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>;
QUERY.IN.SELF: on RENAME_PROVIDER then <policy>;
QUERY.IN.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
QUERY.IN.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
QUERY.SMTX.SELF: on ALTER_SEMANTICS then <policy>;
VIEW.OUT.SELF: on ADD_ATTRIBUTE then <policy>;
VIEW.OUT.SELF: on DELETE_SELF then <policy>;
VIEW.OUT.SELF: on RENAME_SELF then <policy>;
VIEW.OUT.ATTRIBUTES: on DELETE_SELF then <policy>;
VIEW.OUT.ATTRIBUTES: on RENAME_SELF then <policy>;
VIEW.OUT.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
VIEW.OUT.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
VIEW.IN.SELF: on DELETE_PROVIDER then <policy>;
VIEW.IN.SELF: on RENAME_PROVIDER then <policy>;
VIEW.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>;
VIEW.IN.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
VIEW.IN.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
VIEW.SMTX.SELF: on ALTER_SEMANTICS then <policy>;
RELATION.OUT.SELF: on ADD_ATTRIBUTE then <policy>;
RELATION.OUT.SELF: on DELETE_SELF then <policy>;
RELATION.OUT.SELF: on RENAME_SELF then <policy>;
RELATION.OUT.ATTRIBUTES: on DELETE_SELF then <policy>;
RELATION.OUT.ATTRIBUTES: on RENAME_SELF then <policy>;

**Property.** *The list of allowed events is complete*. This can be easily verified by checking the following list of rules against the contents of Table 2.1.

**Customizability**. Whereas our small list of generic, default rules can cover all possible combinations of events and node types, it is quite possible that we want to define different reaction to the same event for different modules. For example, we might wish a certain view to block attribute addition, whereas we would allow another view to adapt to the same event. To facilitate this possibility we allow three layers of rules:

1. General rules about *all* modules and their attributes.
2. Rules that apply to all the attributes of a *specific schema*.
3. Rules that apply to *specific attribute* nodes.

If one observes the syntax of the policy annotation language, all the instantiations of receiver node from (a) to (l) refer to the case of layer 1 (i.e., default rules for all the nodes and modules of the graph), the case (m) refers to all the attributes of a specific schema (layer 2), and, finally, the case (n) refers to individual nodes of the graph (layer 3). In our approach, the semantics of the layers of rules state that *each layer overrides the policy of its previous layers*. This way, if we have a default policy for all relations (layer 1) for a certain event (e.g., rename) we can customize the behavior of a specific relation to be different than the default by defining a specific rule for it (layer 2).

**Reference Example.** Returning to our reference example, the following text represents a set of rules of how policy rules should be written in order to have all nodes of the graph propagating all possible events for all modules, except for V_TR view (in which only the CID attribute will propagate any of its incoming events. The following text covers the 1st set of rules mentioned previously.

```
QUERY.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
QUERY.OUT.SELF: on DELETE_SELF then PROPAGATE;
QUERY.OUT.SELF: on RENAME_SELF then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
QUERY.IN.SELF: on DELETE_PROVIDER then PROPAGATE;
QUERY.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
QUERY.IN.SELF: on RENAME_PROVIDER then PROPAGATE;
QUERY.IN.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
QUERY.IN.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
```

QUERY.SMTX.SELF: on ALTER_SEMANTICS then PROPAGATE;
VIEW.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
VIEW.OUT.SELF: on DELETE_SELF then PROPAGATE;
VIEW.OUT.SELF: on RENAME_SELF then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
VIEW.IN.SELF: on DELETE_PROVIDER then PROPAGATE;
VIEW.IN.SELF: on RENAME_PROVIDER then PROPAGATE;
VIEW.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
VIEW.IN.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
VIEW.IN.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
VIEW.SMTX.SELF: on ALTER_SEMANTICS then PROPAGATE;
RELATION.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
RELATION.OUT.SELF: on DELETE_SELF then PROPAGATE;
RELATION.OUT.SELF: on RENAME_SELF then PROPAGATE;
RELATION.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
RELATION.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;

Assuming that the user wanted for V_TR view to have BLOCK policy, the following text describes the set or rules needed to be written under the general rules that we saw just above.

V_TR_OUT.SELF: on ADD_ATTRIBUTE then BLOCK;
V_TR_OUT.SELF: on DELETE_SELF then BLOCK;
V_TR_OUT.SELF: on RENAME_SELF then BLOCK;
V_TR_OUT.ATTRIBUTES: on DELETE_SELF then BLOCK;
V_TR_OUT.ATTRIBUTES: on RENAME_SELF then BLOCK;
V_TR_OUT.ATTRIBUTES: on DELETE_PROVIDER then BLOCK;
V_TR_OUT.ATTRIBUTES: on RENAME_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.SELF: on DELETE_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.SELF: on RENAME_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.SELF: on ADD_ATTRIBUTE_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.ATTRIBUTES: on DELETE_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.ATTRIBUTES: on RENAME_PROVIDER then BLOCK;
V_TR_IN_V_TR.SELF: on DELETE_PROVIDER then BLOCK;
V_TR_IN_V_TR.SELF: on RENAME_PROVIDER then BLOCK;

V_TR_IN_V_TR.SELF: on ADD_ATTRIBUTE_PROVIDER then BLOCK;
V_TR_IN_V_TR.ATTRIBUTES: on DELETE_PROVIDER then BLOCK;
V_TR_IN_V_TR.ATTRIBUTES: on RENAME_PROVIDER then BLOCK;
V_TR_SMTX.SELF: on ALTER_SEMANTICS then BLOCK;

Finally, the user decided that the CID attribute of the output schema of the V_TR module should have *again* different policy (PROPAGATE instead of BLOCK that was set in the previous set of rules). This is achieved by the following set of policies:

V_TR_OUT.CID: on DELETE_SELF then PROPAGATE;
V_TR_OUT.CID: on RENAME_SELF then PROPAGATE;
V_TR_OUT.CID: on DELETE_PROVIDER then PROPAGATE;
V_TR_OUT.CID: on RENAME_PROVIDER then PROPAGATE;

# Chapter 3

# Propagating the impact of an event over the architecture graph

---

The main goal of our method is to assess the impact of a hypothetical event over an architecture graph. As we have seen, we allow the involved stakeholders to regulate the effect of an event by annotating modules and their components with policies that can either (a) BLOCK the event (requesting thus that the module remains structurally and semantically immune to the tested change), or, (b) accept to adapt to the change and PROPAGATE the event to both the module's internal structure and to subsequent data consumers. In the context of this goal, two tasks are possible, and specifically, (a) the assessment of the impact of the hypothetical change over the architecture graph (i.e., the detection of which modules are affected by the change), and, (b) the rewriting of the graph's modules to adapt to the applied change. In this Chapter, we cover the first of the two tasks and provide a method that highlights which nodes can be affected by a hypothetical event.

We follow [PVSV09, PVS11] in our approach and exploit the nature of the graph's summary at the module level. We want to enforce the principle of *locality independence* which states that each *module receives all the information necessary to assess the impact of an event from its environment (i.e., the other modules in the architecture graph) and decides its reaction and possible adaptation on its own (a.k.a., locally, independently)*. In other words, we would like to visit each module once, provide it with all the information necessary on what has happened to its environment of related modules and let it decide on its reaction without interacting with other modules or some form of central management.

Implementing the principle of locality independence is feasible. As already shown in previous work ([PVSV09, PVS11]), due to the nature of view and query definitions, at the high level, the nodes of the graph's summary form a directed acyclic graph of provider-consumer

dependencies. Thus, it is straightforward to obtain a topological sorting of the summary of the architecture graph. We can easily enforce the rule that modules communicate with each other via a single means: the output schema of a provider module notifies the input schema of a consumer module. In such cases, the following protocol is used:

(i) We topologically sort the graph at the module level.

(ii) We visit each affected module with its topological order and we process there all the incoming messages it has. If this is the case, the topological sort guarantees that all messages pending for the input schemata of the module are ready.

(iii) Every module processes locally the incoming events and also, locally decides the status for its semantics and output schema. Next, it is ready to propagate this information to all its consumers (if any).

In this chapter, we will first start with the description of our method at a high, inter-module level; i.e., we will first present the first two steps of the aforementioned strategy. Then, we will delve into the details of intra-module reaction to change and see how a module reacts to an incoming event internally.

## 3.1 Inter-module propagation of events

In this section, we will present the high-level behavior of our method at the module level. In a nutshell, our algorithm works as follows.

(i) Before any event is to be tested, we topologically sort the modules of the architecture graph. This is performed once, in advance of any impact assessment. The topological sort starts from the relations (that are the fountains of data provision) and moves on to views and queries (that are the intermediaries and targets of data consumption).

(ii) Whenever a hypothetical event is assessed we start from the module over which it is assessed and visit the rest of the nodes by following the topological sorting of the modules. This way, we can ensure that a module is visited after *all* of its data providers have been visited. The visited node assesses the impact of the event internally (see the following section for this) and obtains a *status*, which can be only one of the following:

- BLOCK, meaning that the module is requesting that it remains structurally and semantically immune to the tested change; this also means that the module will block the event (as its immunity obscures the event from its data consumers), or,

- PROPAGATE, meaning that the modules concedes to adapt to the change and propagate the event to any subsequent data consumers.
- NO_STATUS, meaning that the module is not affected by the change.

(iii) If the status of the module is PROPAGATE, the event must be propagated to the subsequent modules. To this end, the visited module prepares *messages* for its data consumers, notifying them about its own changes. These messages are pushed to a central, common message queue (where messages are sorted by their target module's topological sorting identifier).

(iv) The process terminates whenever there are no more messages and no more modules to be visited.

In this section, we start our discussion with the topological sorting of the graph's summary at the module level. Second, we present the algorithm for the assessment of which nodes are affected by a hypothetical change. Then, we give some implementation details on the messages that are used to allow the communication of modules.

### 3.1.1 Topological sorting

We perform a topological sort of high-level nodes (Queries, Views, Relations) before doing any step that can cause changes to our graph. This way we are sure that the messages that are transferred between high-level nodes are transferred in the right order from providers to consumers. This sorting is done by setting incremental numbers at nodes, called IDs. At the end of the algorithm, all high level nodes have a unique ID. Relations have the smallest IDs, followed by Views and Queries.

We follow a traditional approach to our topological sorting, which proceeds as follows: first we find the nodes with zero incoming edges. These nodes are removed from the examination set with their outgoing edges (uses edges). This gives as a result a new set of nodes with zero incoming edges. The algorithm stops when there are no more nodes to visit.

**Algorithm 1** Topological sorting algorithm

**Input:** A summary of an architecture graph $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$ that comprises the modules of an architecture graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$

**Ouput:** A topologically sorted architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, i.e. an annotation of the nodes of $\mathbf{G_s}$ with a sequential id's, via a mapping $Y : \mathbf{V_s} \rightarrow \mathbb{N}$

1: **Begin**
2: $notYetVisited = (Relation \cup View \cup Query)$;
3: $algoID =$ size($notYetVisited$);
4: $node = null$;
5: **while** $notYetVisited > 0$ **do**
6: find $node$ with 0 incoming edges from $notYetVisited$;
7: remove $node$ from $notYetVisited$;
8: remove edges starting from $node$;
9: $node.ID = algoID$;
10: $algoID = algoID - 1$;
11: **end while**
12: **End**

Returning to our reference example the above algorithm of topological sorting produces the following output:

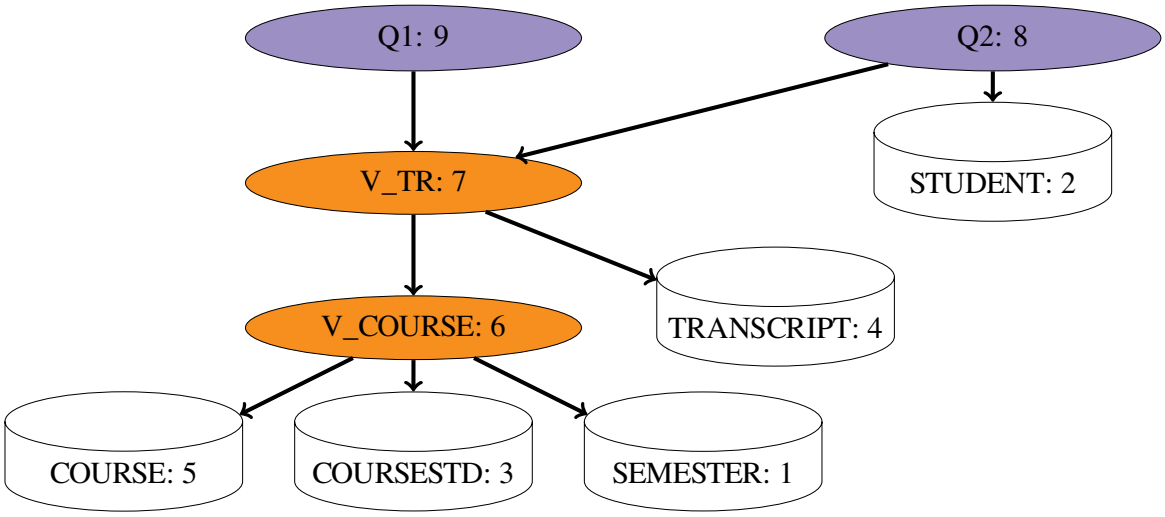



**Figure 3.1:** Summary of the architecture graph of the reference example with the nodes annotated with the id obtained by the topological sorting of the architecture graph

### 3.1.2 General algorithm at module level and explanation of queue

Once the graph is topologically sorted, we exploit this order to propagate an event through it. Whenever the user initiates a hypothetical event on a graph's node, we start from the affected node's module and visit each module in the order dictated by the topological sorting algorithm. Each visited module has to check what its policy towards this event is and, if the policy is to further propagate the event, notify its consumer modules for the change. To facilitate the module level communication, we use a $global\ queue$ of messages that is sorted with respect to the topological order of the modules. This allows us to use the following protocol of communication:

**Algorithm 2** Status determination algorithm

**Input:** A topologically sorted architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$ (output of algorithm 1), a global queue $Q$ that facilitates the exchange of messages between modules

**Ouput:** A list of modules $Affected\ Modules \subseteq \mathbf{V_s}$ that were affected by the event and acquire a status other than $NO\_STATUS$

```
1:  function SetStatus(Module, Messages)
2:      Consumers Messages = ∅;
3:      for all Message ∈ Messages do
4:          decide status of Module;
5:          put messages for Module's consumers in Consumers Messages;
6:      end for
7:  end function
8:  Begin
9:      for all node ∈ G_s(V_s, E_s) do
10:         node.status = NO_STATUS;
11:     end for
12:     while size(Q) > 0 do
13:         visit module (node) in head of Q;
14:         insert node in Affected Modules list;
15:         get all messages, Messages, that refer to node;
16:         SetStatus(node, Messages);
17:         if node.status == PROPAGATE then
18:             insert node.Consumers Messages to the Q;
19:         end if
20:     end while
21:     return Affected Modules;
22: End
```

Whenever a module is processed by the algorithm it receives *all* messages that pertain to this module. This happens because there can be more than one messages for a module in the $global\ queue$. For example, with the deletion of an attribute that was used in the output schema of a module and in the semantics schema of a module, the module should inform its consumers that (a) the attribute was deleted and (b) its semantics has changed. If $node.status == block$, then the messages that were produced for $node$ consumers will not be inserted in the global queue. If there are still messages in the queue, these messages will be processed by the *Status Determination*. In our reference example, we can see that there are two final destinations (the two queries) from a event that initiated in V_TR module, for example. This means that even if one of Q1 or Q2 sets its status as block, the other might want to accept the imminent change.

Having discussed the body of the algorithm, we now turn our attention to some implementation details on the structure and recipients of the emitted messages.

### 3.1.3 Message structure and content

Each message $msg$ is a quadruple $msg(n, s, e, p)$ with the following parts:

- a recipient module $n$ that is the target of the message, to be used in order to place the message appropriately in the common message queue, which is sorted by the id of the modules (as dictated by the topological sorting).
- the specific schema $s$ of $n$, to which the message is sent (note that due to this information, we can also find who the sender of the message was, since an input schema has exactly one provider)
- the event $e$ that this message carries.
- any parameters $p$ containing additional information needed for some events (e.g., addition or renaming events).

Since the user can select only one event to take place, in the beginning there is going to be only one initial message. Depending on the user selection this message should carry among nodes (to node and to schema) and needed parameter the following event information:

**DELETE_ATTRIBUTE:** when the user wants to delete one attribute from the output schema (the user selects DELETE_SELF on an attribute but on initialization, since we know that the user selected an attribute and not a module we change the event from DELETE_SELF to DELETE_ATTRIBUTE).

**RENAME_ATTRIBUTE:** when the user wants to rename one attribute from the output schema (likewise the user selects RENAME_SELF on an attribute but on initialization we change that from RENAME_SELF to RENAME_ATTRIBUTE).

**ADD_ATTRIBUTE:** when the user wants to add another attribute to the output schema of a module.

**DELETE_SELF:** when the user wants to delete a module.

**RENAME_SELF:** when the user wants to rename a module.

**ALTER_SEMANTICS:** when the user wants to change the semantics of a module.

Depending on the type of event, when once the module has determined its reaction, it constructs messages for its data consumers. Here, we list some examples of such cases.

- When a message is processed saying that an attribute is going to be deleted, the input schema of the consumers that are connected to that attribute is informed that the attribute will be deleted.
- If the whole module is going to be deleted then the consumers of this module will receive a message in their input schema saying that the provider of that input schema is going to be deleted.
- Likewise when an attribute is going to be renamed, the input schema of the consumers that are connected to that attribute is informed that the attribute will have a new name from now on.
- If the whole module is going to be renamed, then the consumers of this module will receive a message in their input schema saying that the provider of that input schema is going to be renamed.
- When a module processes a message saying that a new attribute is going to be added to its output schema, it informs all of its consumers in their input schema that a new attribute was added to their provider.
- Finally when a module processes a message saying that its semantics have changes, it informs all its consumers that it changed its semantics.

## 3.2 Intra-module processing of arriving events

As already mentioned, a guiding principle of our design is the principle of locality independence. The crux of the principle is that as events flow over the graph, each module (a) assesses its reaction and (b) adapts to the changes, *independently*. To allow the smooth scale-up of our algorithm concerning the size of the architecture graph, we wish to avoid having any central mechanism to dictate what each module does; on the contrary, we wish that each module can process incoming events without negotiating with other modules or central mechanisms. We exploit the topological ordering of the graph to make sure that whenever a module is to be processed, *all* the messages that concern it have been delivered to the message queue. Therefore, the module retrieves all these messages from the common queue and processes them internally. In the rest of this section, we discuss how this processing takes place. However, before proceeding, we stop to make a few comments with observations and explanations that will allow the better understanding of our technical decisions and methods.

**Algorithmic steps.** Independently of the type of event, or the kind of module that receives a message, the main algorithmic steps of the module in order to assess its reaction to the incoming message are as follows.

1. Whenever visited, a module starts by retrieving from the common queue all the messages that concern it. Then, it processes all these messages sequentially, under the following order of actions.

2. First, the module probes its schemata for their reaction to the incoming event, starting from the input schemata, next to the semantics and finally to the output schema. Naturally, relations deal only with the output schema.

3. Within each schema, the schema has to probe both itself and its contained nodes (attributes)[1] for their reaction to the incoming event. At the end of this process (to be detailed in a case by case mode right next) the schema assumes a status

4. Once all schemata have assumed status, it is the output schema of the module that decides the reaction of the overall module; if any of the schemata raises a veto (BLOCK) the module assumes the BLOCK status too; otherwise, it assumes the PROPAGATE status

5. Finally, if the module assumes the PROPAGATE status, it prepares messages for all its consumers and inserts them in the common queue.

Interestingly, within each module, the schemata communicate also with a local queue. Also, it is noteworthy that every time a module is visited, it is possible that there are more than

[1]For reasons of efficiency, in our implementation we have simplified the reaction of the semantics schema to avoid examining its constituents; however, in principle, this is feasible too

one messages that concern it. Typically this is due to one of the following two phenomena (not excluding other possibilities): (a) cases of self-join, where a provider feeds (directly or indirectly) a consumer with more than one paths (and thus, a change in the provider concerns more than one schemata of the consumer – observe here that it is not obligatory that these schemata have identical reaction towards the event) and (b) a deletion of an attribute in a view might affect both the semantics and the output schema of the view, producing thus, two messages to its consumers: one that notifies them that the set of attributes that was exported to them has changed and another notifying them that the semantics of the view has changed (e.g., a part of the SELECT clause has been dropped due to the attribute deletion).

**Maestros for the local processing.** To facilitate the local, independent nature of message processing by the modules, each module awakes a maestro that handles the probing of schemata as well as the decision making on what status will the schema assume. A maestro is a simple piece of software (implemented as an abstract interface, later materialized on a case by case basis) that is specialized on the combination $type\,of\;event\;\times\;module\,type$. For each type of module, there is a specialized maestro that takes care of status determination for each possible event that can be received.

In terms of software architecture, the decision for this structuring of the code was done in order to decentralize event processing. It allows the reasonably smooth extension of the architecture with new types of events or modules at the price of some code reusability. In terms of algorithmic principles, we gain the benefits of module independence at the price of a common queue of messages.

In the following, we present how events are processed inside modules, organized by the type of the incoming message that the module is called to handle. For each event, we explain the structure of the incoming message and the list of steps that have to take place (organized per schema, if more than one schemata of the module are involved).

### 3.2.1 Intra-module processing at relation modules

In this subsection, we discuss how events are processed inside relation modules.

- Incoming message for **attribute deletion**.

| To node | Relation |
|---|---|
| To schema | Relation output schema |
| Event | DELETE_ATTRIBUTE |
| Parameters | Attribute name (*AN*) |

  1. Probing the output schema.

(a) Output schema searches for the attribute that is named as *AN* (denoted as *R.O.A*).

(b) *R.O.A* policy is checked for deletion, if *R.O.A* has block policy then *R.O.A* and output schema statuses are set to *BLOCK*, otherwise statuses are set to *PROPAGATE*.

(c) Finally if output schema status is *PROPAGATE* a new set of messages is inserted in the local queue of outgoing messages for this module's consumers, these messages say that this module deleted this (*R.O.A*) attribute.

- Incoming message for **attribute renaming**.

| To node | Relation |
|---|---|
| To schema | Relation output schema |
| Event | RENAME_ATTRIBUTE |
| Parameters | Attribute name (*AN*) |

1. Probing the output schema.

   (a) Output schema searches for the attribute that is named as *AN* (denoted as *R.O.A*).

   (b) *R.O.A* policy is checked for renaming, if *R.O.A* has block policy then *R.O.A* and output schema statuses are *BLOCK*, otherwise statuses are set to *PROPAGATE*.

   (c) Finally if output schema status is assumed *PROPAGATE* a new set of messages is inserted in the local queue of outgoing messages for this module's consumers, these messages say that this module renamed this (*R.O.A*) attribute.

- Incoming message for **self deletion**.

| To node | Relation |
|---|---|
| To schema | Relation output schema |
| Event | DELETE_SELF |
| Parameters | |

1. Probing the output schema.

   (a) Output schema checks for each one of its attributes policy for deletion, if any has block policy then that attribute and output schema get status *BLOCK*, otherwise status *PROPAGATE* is set to all (attributes and output schema).

   (b) Finally if output schema status is assumed *PROPAGATE* a new set of messages is inserted in the local queue of outgoing messages for this module's consumers, these messages say that this module is deleted.

- Incoming message for: **self renaming**.

| To node | Relation |
|---|---|
| To schema | Relation output schema |
| Event | RENAME_SELF |
| Parameters | |

1. Probing the output schema.
   (a) Output schema checks its policy for renaming, if it has block policy then output schema gets status *BLOCK*, otherwise its status is set to *PROPAGATE*.
   (b) Finally if output schema status is assumed *PROPAGATE* a new set of messages is inserted in the local queue of outgoing messages for this module's consumers, these messages say that this module is renamed.

- Incoming message for **attribute addition**.

| To node | Relation |
|---|---|
| To schema | Relation output schema |
| Event | ADD_ATTRIBUTE |
| Parameters | |

1. Probing the output schema.
   (a) Output schema checks its policy for adding attributes, if it has block policy then output schema gets status *BLOCK*, otherwise its status is set to *PROPAGATE*.
   (b) Finally if output schema status is assumed *PROPAGATE* a new set of messages is inserted in the local queue of outgoing messages for this module's consumers, these messages say that this module has added a new attribute.

### 3.2.2 Intra-module processing at query or view modules

In this subsection, we discuss how events are processed inside query or view modules.

- Incoming message for **provider attribute deletion**.

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | DELETE_PROVIDER |
| Parameters | Attribute name (*AN*) |

1. Probing the input schema.

(a) Input schema searches for the attribute named as *AN* (denoted as *Q.I.A*).

(b) *Q.I.A* policy is checked for deletion, if *Q.I.A* has block policy then *Q.I.A* and input schema statuses are set to *BLOCK*, otherwise both nodes assume status *PROPAGATE*.

2. Probing the semantics schema.

(a) Semantics schema searches if it has connection with the attribute *Q.I.A*.

(b) If semantics schema has connection with *Q.I.A* then semantics policy is checked for alteration, if it has block policy then semantics schema status is set to *BLOCK*, otherwise it is set to *PROPAGATE*.

(c) Finally if semantics schema has status *PROPAGATE* then a new set of messages is inserted in the local queue of outgoing messages for this modules consumers that say that this module has changed its semantics.

3. Probing the output schema.

(a) Output schema searches for the attributes that have connection with *Q.I.A* (denoted as *Q.O.As*).

(b) If none *Q.O.A* exists, output schema stops further execution for this message.

(c) If any *Q.O.A* exists, then for each *Q.O.A* its policy is checked for deletion, if any of the *Q.O.As* has block policy then this *Q.O.A* and output schema assume *BLOCK* status otherwise their statuses are set to *PROPAGATE*.

(d) Finally if output schema has status *PROPAGATE*, then for each *Q.O.A* a new set of messages is inserted in the local queue of outgoing messages for this modules consumers that say that this module deleted this (*Q.O.A*) attribute.

- Incoming message for **provider schema deletion**.

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | DELETE_PROVIDER |
| Parameters | Schema name (*SN*) |

1. Probing the input schema.

(a) For each one of the attributes of input schema (denoted as *Q.I.As*) policy is checked for deletion, if any of *Q.I.A* has block policy then *Q.I.A* and input schema statuses are set to *BLOCK*.

(b) If all *Q.I.As* statuses are *PROPAGATE* then the policy of the input schema is checked for deletion. If it has block policy then the status of input schema is set to *BLOCK* else it is set to *PROPAGATE*.

2. Probing the semantics schema.
   (a) Semantics schema searches if it has connection with any of the attributes *Q.I.As*.
   (b) If semantics schema has connection with a *Q.I.A* then semantics policy is checked for alteration, if it has block policy then semantics schema status is set to *BLOCK*, otherwise it is set to *PROPAGATE*.
   (c) Finally if semantics schema has status *PROPAGATE* then a new set of messages is inserted in the local queue of outgoing messages for this modules consumers that say that this module has changed its semantics.
3. Probing the output schema.
   (a) Output schema searches for the attributes that have connection with *Q.I.As* (denoted as *Q.O.As*).
   (b) If none *Q.O.A* exists, output schema stops further execution for this message.
   (c) If any *Q.O.A* exists, then for each *Q.O.A* its policy is checked for deletion, if any of the *Q.O.As* has block policy then this *Q.O.A* and output schema assume *BLOCK* status otherwise their statuses are set to *PROPAGATE*.
   (d) Finally if output schema has status *PROPAGATE*, then for each *Q.O.A* a new set of messages is inserted in the local queue of outgoing messages for this modules consumers that say that this module deleted this (*Q.O.A*) attribute.

- Incoming message for **attribute deletion**.

| To node | Query/View |
|---|---|
| To schema | Query/View output schema |
| Event | DELETE_ATTRIBUTE |
| Parameters | Attribute name (*AN*) |

1. Probing the output schema.
   (a) Output schema searches for the attribute that is named as *AN* (denoted as *Q.O.A*).
   (b) *Q.O.A* policy is checked for deletion, if *Q.O.A* has block policy then *Q.O.A* and output schema statuses are set to *BLOCK*, otherwise statuses are set to *PROPAGATE*.
   (c) Finally if output schema status is *PROPAGATE* a new set of messages is inserted in the local queue of outgoing messages for this module's consumers, these messages say that this module deleted this (*Q.O.A* attribute.

- Incoming message for **provider attribute renaming** (generated by a provider).

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | RENAME_PROVIDER |
| Parameters | Attribute name (*AN*) |

  1. Probing the input schema.
     (a) Input schema searches for the attribute named as *AN* (denoted as *Q.I.A*).
     (b) *Q.I.A* policy is checked for renaming, if *Q.I.A* has block policy then *Q.I.A* and input schema statuses are set to *BLOCK*, otherwise both nodes assume status *PROPAGATE*.
  2. Probing the output schema.
     (a) Output schema searches for the attributes that have connection with *Q.I.A* and are named as *AN* (denoted as (*Q.O.A*).
     (b) If *Q.O.A* does not exist, output schema stops further execution for this message.
     (c) If *Q.O.A* exists, then we check *Q.O.A's* policy for renaming, if it has block policy then this *Q.O.A* and output schema status are set to *BLOCK*, otherwise their statuses are set to*PROPAGATE*.
     (d) Finally a new set of messages is inserted in the local queue of outgoing messages for this modules consumers that say that this module renamed this (*Q.O.A*) attribute.

- Incoming message for **provider schema renaming**.

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | RENAME_PROVIDER |
| Parameters | Schema name (*SN*) |

  1. Probing the input schema.
     (a) Input schema policy is checked for deletion, if it has block policy then input schema status is set to *BLOCK*, else it is set to *PROPAGATE*.

- Incoming message for **attribute renaming**.

| To node | Query/View |
|---|---|
| To schema | Query/View output schema |
| Event | RENAME_ATTRIBUTE |
| Parameters | Attribute name (*AN*) |

1. Probing the output schema.
   (a) Output schema searches for the attribute that is named as *AN* (denoted as *Q.O.A*).
   (b) *Q.O.A* policy is checked for renaming, if *Q.O.A* has block policy then *Q.O.A* and output schema statuses are *BLOCK*, otherwise statuses are set to *PROPAGATE*.
   (c) Finally if output schema status is assumed *PROPAGATE* a new set of messages is inserted in the local queue of outgoing messages for this module's consumers, these messages say that this module renamed this (*Q.O.A*) attribute.

- Incoming message for **self deletion**.

| To node | Query/View |
|---|---|
| To schema | Query/View output schema |
| Event | DELETE_SELF |
| Parameters | |

1. Probing the output schema.
   (a) Output schema checks for each one of its attributes policy for deletion, if any has block policy then that attribute and output schema get status *BLOCK*, otherwise status *PROPAGATE* is set to all (attributes and output schema).
   (b) Finally if output schema status is assumed *PROPAGATE* a new set of messages is inserted in the local queue of outgoing messages for this module's consumers, these messages say that this module is deleted.

- Incoming message: **self renaming**.

| To node | Query/View |
|---|---|
| To schema | Query/View output schema |
| Event | RENAME_SELF |
| Parameters | |

1. Probing the output schema.
   (a) Output schema checks its policy for renaming, if it has block policy then output schema gets status *BLOCK*, otherwise its status is set to *PROPAGATE*.
   (b) Finally if output schema status is assumed *PROPAGATE* a new set of messages is inserted in the local queue of outgoing messages for this module's consumers, these messages say that this module is renamed.

- Incoming message for **provider attribute addition**.

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | ADD_ATTRIBUTE_PROVIDER |
| Parameters | |

  1. Probing the input schema.
     (a) Input schema checks for its policy for adding attribute, if it has block policy then it sets its status to *BLOCK*, otherwise its status is set to *PROPAGATE*.
  2. Probing the semantics schema.
     (a) Semantics schema checks for its policy for alter semantics, if it has block policy then semantics schema get status *BLOCK*, otherwise its status is set to *PROPAGATE*.
     (b) Finally if semantics schema has status *PROPAGATE* then a new set of messages is inserted in the local queue of outgoing messages for this modules consumers that say that this module has changed its semantics.
  3. Probing the output schema.
     (a) Output schema checks for its policy for adding attributes, if it has block policy then output schema get status *BLOCK*, otherwise its status is set to *PROPAGATE*.
     (b) Finally if output schema has status *PROPAGATE* then a new set of messages is inserted in the local queue of outgoing messages for this modules consumers that say that this module has added an attribute.

- Incoming message for **attribute addition**.

| To node | Query/View |
|---|---|
| To schema | Query/View output schema |
| Event | ADD_ATTRIBUTE |
| Parameters | |

  1. Probing the output schema.
     (a) Output schema checks for its policy for adding attribute, if it has block policy then output schema get status *BLOCK*, otherwise its status is set to *PROPAGATE*.

(b) Finally if output schema has status *PROPAGATE* then a new set of messages is inserted in the local queue of outgoing messages for this modules consumers that say that this module has added an attribute.

- Incoming message for **alter semantics** (user generated).

| To node | Query/View |
|---|---|
| To schema | Query/View semantics schema |
| Event | ALTER_SEMANTICS |
| Parameters | |

1. Probing the semantics schema.
   (a) Semantics schema checks for its policy for altering its semantics, if it has block policy then semantics schema get status *BLOCK*, otherwise its status is set to *PROPAGATE*.
   (b) Finally if semantics schema has status *PROPAGATE* then a new set of messages is inserted in the local queue of outgoing messages for this modules consumers that say that this module has changed its semantics.

- Incoming message for **alter semantics** (system generated).

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | ALTER_SEMANTICS |
| Parameters | |

1. Probing the semantics schema.
   (a) Semantics schema checks for its policy for altering its semantics, if it has block policy then semantics schema get status *BLOCK*, otherwise its status is set to *PROPAGATE*.
   (b) Finally if semantics schema has status *PROPAGATE* then a new set of messages is inserted in the local queue of outgoing messages for this modules consumers that say that this module has changed its semantics.

## 3.3 Theoretical guarantees

The theoretical foundations and guarantees for this chapter have been laid out in [PVS11]. We adapt the proofs to our implementation in this section. We also refer the interested reader to the following section for a discussion of similarities and differences with [PVS11].

### 3.3.1 Termination and confluence at inter-module level

First, we prove that the mechanism for message propagation works correctly at the inter-module level.

*Theorem 1 (termination).* The message propagation at the inter-module level terminates.

*Proof*: The summary of the architecture graph is a directed acyclic cycle. This is due to the fact that a query depends only on views and relations and relations do not depend on anything (in the context of this paper, we do not consider cyclic foreign key dependencies). Since the summary graph is a DAG, we can topologically sort it and propagate the messages according to this topological order. Thus, all that it takes for the message propagation mechanism to terminate is: (a) each module emits message only once for each session to its consumers that are related with the event/parameter; (b) the graph is finite. Since both assumptions hold, the algorithm terminates.

*Theorem 2 (unique status).* Each module in the graph will assume a status once the message propagation terminates.

*Proof*: Each module gathers from the common message queue all the messages that concern it. For each message, the module and its schemata, assume a status. A module's status can change only in the following order: NO_STATUS < PROPAGATE < BLOCK, meaning that if a module has assumed PROPAGATE status, it can not change it to NO_STATUS but it may change it to BLOCK. At the end of the message processing, the module retains the final status it assumed.

*Theorem 3 (correctness).* Messages are correctly propagated to the modules of the graph.

*Proof*: The modules that must be appropriately notified are these for which an event occurs at their providers. By definition, at the summary level, the architecture graph is a connected graph, where one (or more) input schema node(s) of a consumer module is connected via directed edges to the output schema node(s) of its providers. The messaging mechanism dictates that each message is propagated from the output node of the provider module towards the input schema node of all consumer modules, unless a block policy explicitly halts the propagation. Thus, the connectivity of the graph assures that the modules, which are eventually visited by the message propagation mechanism, have at least one of their providers affected. On the other hand, the modules that are not visited by the mechanism (a) either do not have any provider affected, or, (b) a block policy exists.

### 3.3.2 Termination and confluence at intra-module level

*Theorem 4 (termination and correctness).* The message propagation at the intra-module level terminates and each node assumes a unique status according to its policy and the status precedence constraints.

*Proof*: We visit the schemata of a module in a fixed order: input schema, semantics schema, output schema. For each of these schemata, we may visit its attributes. All these constructs are finite and visited only once. Therefore, the algorithm terminates. Every time a schema is probed on an event, (depending on the event) (a) the appropriate nodes within a schema are probed (depending on the event), and, (b) the schema itself is also prompted. If a more detailed construct is affected, it overrides the policy of the schema in all occasions. For cases where only the schema should be prompted (e.g. attribute addition) we do not interfere with its attributes. This is the correct and desired behavior. The completeness of the language guarantees that *all* nodes have a policy for any incoming event that can arrive to them. Therefore, in all occasions (a) the correct nodes are prompted for a response, (b) the policy of the appropriate nodes prevails, (c) it is impossible that such a policy does not exist. Therefore, for each message all nodes acquire the correct status. For the case of multiple messages in the same module, as already mentioned, vetoes override propagation. If a module has received a message (e.g. delete attribute) for which it has responded with BLOCK and later it processes a second message (e.g. alter semantics) for which it responds with PROPAGATE, it will be the BLOCK status that will prevail in the end. Overall:

- *per message*, all the appropriate nodes (and only them) are visited exactly once and the status of the most detailed nodes overrides the decision of the status of the schema
- *for all messages*, vetoes override adaptation and thus the status acquisitions is confluent
- if any of the schemata of a module has status BLOCK, the module assumes status BLOCK.

## 3.4 Implementation and comparison to state of the art

The main ideas that have been implemented int the context of this thesis have first appeared in [PVS11]. Following the proposal of [PVS11], in our intra-module processing of the messages, we start by checking the policy for the event of nodes at the input schema, then we proceed by checking if there is any connection to the semantics schema, in order to check its policy for the event and finally we check if there exist connections to the output schema, so as to check the policy about the event, there too.

However, the concrete implementation of the ideas proposed in [PVS11] led to several differences and improvements that are listed below:

1. We let modules to send more than one message to their consumers. The authors of [PVS11] proposed that a module should obtain one status containing the information of

all of its incoming messages and after status determination the module would send to its receivers only one message containing that information.

2. Instead of using a big number of statuses (equal to the number of events $\times$ number of provider nodes), we use only 2 statuses (PROPAGATE and BLOCK) along with the incoming messages of the module. This combination produces the same number of statuses as the ones in [PVS11].

3. The policies are written in a simpler way. Previously, the user had to write 4 parts on a policy sentence (the provider node of the event, the receiver node of the event, the event and finally the acceptance or declination of the event). Now the user has to write 3 parts on a policy sentence (this happens because we make a better use of the new model of the architecture graph, since each receiver node has only one provider).

4. Messages are constructed only for the module's consumers that are related to the node that accepts the event, for example a deletion of attribute on a relation module that no-one uses, produces no message at all. Before inserting the produced messages in the global queue we check the status of the module. If it is PROPAGATE we insert the messages to the queue, otherwise we ignore them.

5. We do not visit all modules of the graph that are topologically sorted after the module that initialized the event to see if it has messages. We use the global queue of messages to see what module we will visit next. Since our global queue is sorted with respect to the topological sort, and, as previously said, we do not produce messages for all but only for the affected modules, we can reduce the number of messages to only the necessary ones.

6. As previously mentioned, for reasons of efficiency, we have simplified the reaction of the semantics schema to avoid examining its constituents. In other words, we do not set policies for each node of the semantics tree individually, but rather we set a single policy for the entire tree. However, this extension is clear feasible in the same mentality we have treated the other schemata.

# Chapter 4

# Query and view rewriting to accommodate change in the architecture graph

---

In Chapter 3, we have described how we can identify which parts of the architecture graph are affected by a hypothetical event. Our method works as follows: we follow each module (relation/view/query) of the graph in the order dictated by the topological sort with the aim to determine in which way the module is affected by the propagation of the event. Internally, the module checks the policies of its component schemata and based on these policies it adopts a status and notifies its consumers if necessary.

If however, we wish not only to identify affected nodes, but also to present the user with a view of how the architecture graph will look like if the hypothetical event is actually applied over the ecosystem, there are three steps in order for the adaptation of the architecture graph to take place:

1. Status Determination
2. Path Check
3. Graph Rewrite

Our goal is to rewrite the architecture graph in order to reflect the response of the modules to the events that arrive to them. The first step of the method, Status Determination, has been already covered in Chapter 3. Then, the problem would intuitively seem simple: the status determination algorithm has determined the status of each module; then each module gets rewritten if the status is PROPAGATE and remains the same if the status is BLOCK. This would require only the execution of the Graph Rewrite step – in fact, one could envision cases where

Status Determination and Graph Rewrite could be combined in a single pass. Unfortunately, although the decision on Status Determination can be made locally in each module, taking into consideration only the events generated by previous modules and the local policies, the decision on rewriting has to take extra information into consideration. This information is not local, and even worse, it pertains to the subsequent, consumer modules of an affected module, making thus impossible to weave this information in the first step of the method, Status Determination. Observe the following example:

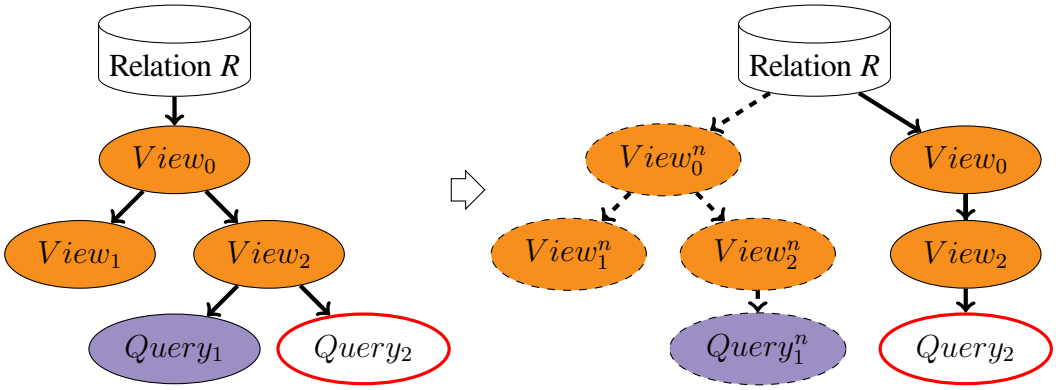



**Figure 4.1:** Block rewriting example

In the example of Figure 4.1, we have a relation *R* and a view $View_0$ that gets data from the relation. Two views ($View_1$ and $View_2$) use $View_0$ in order to get data. $View_2$ is further used by two queries ($Query_1$ and $Query_2$). The database administrator wants to change $View_0$, in a way that all modules depending on $View_0$ are going to be affected by that change (e.g., attribute deletion, for an attribute common to all the modules of the example). Assume now that all modules except $Query_2$ accept to adapt to the change, as they have a PROPAGATE policy annotation. $Query_2$ module that vetoed must be kept immune to the change; to achieve this we must retain the previous version of *all* the nodes in the path from the origin of the evolution ($View_0$) to the blocking $Query_2$. As one can see in the figure, we now have two variants of $View_0$ and $View_2$: the new ones (named $View_0^n$ and $View_2^n$) that are adapted to the new structure of $View_0$ – now named $View_0^n$ – and the old ones, that retain their name and are depicted in the rightmost part of the figure. The latter are immune to the change and their existence serves the purpose of correctly defining $Query_2$.

The crux of the problem is as follows: if a module has PROPAGATE status and none of its consumers, including both the immediate of the transitive consumers (i.e., the consumers of the consumers) raises a BLOCK, then both the module and all of these consumers are rewritten to

a new version.

However, if any of the immediate consumers, or any of the transitive consumers of a module raises a veto, BLOCK status, then *the whole path towards this vetoing node must hold two versions of each module*: (a) the new version, as the module has accepted to adapt to the change by assuming a PROPAGATE status and (b) the old version in order to server the correct definition of the vetoing module.

To correctly serve the above purpose, the second step of our method, Path Check, works from the consumers towards the providers in order to inform each module about its path that it has to retain both its old and a its new version.

The third part of the method visits each module and rewrites the module as follows:

- If the module needs to retain only the new version, we visit that module and perform the needed change, then we connect it correctly to the providers that it should have.
- If the module needs both the old and the new versions, we make a clone of the module to our graph and we perform the needed change over the cloned module, then we connect it correctly to the providers that it should have.
- If the module needs to retain only the old version, we do not perform any change. This happens when the module we are visiting is the one that raised the veto.

The details of the rewriting process are presented in sections 4.2.1 and 4.2.2.

## 4.1 Algorithms for graph rewriting

In the Path Check step of our algorithm, we have already set the statuses of the nodes and we use them for the rewriting of the modules or not. If we have a BLOCK at any of the nodes then we start a reverse traversal of the nodes starting from the blocker modules to the module that initialized the flow and we inform each module in that path that he should keep his present form and create a new version of itself with the change that user wants.

The only exception to rewriting is when the module of the initial message is a relation module.

**Algorithm 3** Path check algorithm

**Input:** A summary of an architecture graph $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, a list of modules $Affected\ modules$, that were affected by the event (output of algorithm 2)

**Ouput:** Annotation of the modules of $Affected\ modules$ on the action needed to take, and specifically whether we have to make a new version of it, or, implement the change the user asked on the current version

```
1:  function CheckModule(Module, Affected modules)
2:      if Module has been marked then
3:          return;                                   ▷ notified by previous block path
4:      end if
5:      mark Module to keep current version and apply the change on a clone;
6:      for all New module ∈ Affected modules feeding Module do
7:          CheckModule(New module, Affected modules);          ▷ notify path
8:      end for
9:  end function
10: Begin
11:     for all Module ∈ Affected modules do
12:         if Module.status == BLOCK then
13:             CheckModule(Module, Affected modules);
14:             mark Module not to change;        ▷ blockers keep only current version
15:         end if
16:     end for
17: End
```

Finally, all nodes that can be rewritten are getting their new definition according to their incoming events as described in relation maestro for rewrite and query/view maestro for rewrite (Sections 4.2.1 and 4.2.2 recursively).

**Algorithm 4** Rewriting algorithm

**Input:** A list of modules $Affected\ modules$, knowing the number of versions they have to retain (output of algorithm 3), initial messages of $Affected\ modules$

**Ouput:** Architecture graph after the implementation of the change the user asked

```
1:  Begin
2:      if any of Affected modules has status BLOCK then
3:          if initial message started from Relation module type then
4:              return ;                                  ▷ Relations do not change at all
5:          else
6:              for all Module ∈ Affected modules do
7:                  if Module needs only new version then
8:                      proceed with rewriting of Module;
9:                      connect Module to new providers;     ▷ new version goes to new path
10:                 else
11:                     clone Module;                        ▷ clone module, to keep both versions
12:                     connect cloned Module to new providers;   ▷ clone is the new version
13:                     proceed with rewriting of cloned Module;
14:                 end if
15:             end for
16:         end if
17:     else
18:         for all Module ∈ Affected modules do
19:             proceed with rewriting of Module                  ▷ no blocker node;
20:         end for
21:     end if
22: End
```

As in the case of Status Determination, the rewriting of graph modules has been implemented via the appropriate *maestros*. Every maestro, is again customized for the combination $type\ of\ event \times\ module\ type$. For each type of module, there is a specialized maestro for the rewriting of the module for each possible event that can be received. The functionality of the rewriting maestros is simpler as they simply have to check with each schema whether a modification of its structure is required, via the addition or deletion of nodes.

## 4.2 Intra-module rewriting mechanism

In the following, we detail the functionality of rewrite maestros, organized by the type of the incoming message that the maestro is called to handle.

### 4.2.1 Relation maestro for rewrite

In this subsection, we discuss how events are processed inside relation modules.

- Initial message for **attribute deletion**.

| To node | Relation |
|---|---|
| To schema | Relation output schema |
| Event | DELETE_ATTRIBUTE |
| Parameters | Attribute name (*AN*) |

  1. Probing the output schema.
     (a) Output schema searches for the attribute that is named as *AN* (denoted as *R.O.A*).
     (b) *R.O.A* is deleted and its name is propagated to other rewrite maestros.

- Initial message for **attribute renaming**.

| To node | Relation |
|---|---|
| To schema | Relation output schema |
| Event | RENAME_ATTRIBUTE |
| Parameters | Attribute name (*AN*) |

  1. Probing the output schema.
     (a) Output schema searches for the attribute that is named as *AN* (denoted as *R.O.A*).
     (b) User is asked for the new name of *R.O.A* and then *R.O.A* is renamed to what user selected if that name is not already present at the output schema of the module, then this name is propagated to other rewrite maestros.

- Initial message for **self deletion**.

| To node | Relation |
|---|---|
| To schema | Relation output schema |
| Event | DELETE_SELF |
| Parameters | |

1. Probing the output schema.
   (a) Output schema gets deleted.
   (b) Module node gets deleted.

- Initial message for: **self renaming**.

| To node | Relation |
|---|---|
| To schema | Relation output schema |
| Event | RENAME_SELF |
| Parameters | |

1. Probing the output schema.
   (a) User is asked for the new name of the module. Then the module is renamed to what the user selected if this module is not already present at the graph and this name is propagated to other rewrite maestros.

- Initial message for **attribute addition**.

| To node | Relation |
|---|---|
| To schema | Relation output schema |
| Event | ADD_ATTRIBUTE |
| Parameters | |

1. Probing the output schema.
   (a) User is asked for the name of the new attribute. When that name is not already present at the output schema of the module, a new attribute is added to the output schema of the module, then this name is propagated to other rewrite maestros.

### 4.2.2 Query/View maestro for rewrite

In this subsection, we discuss how events are processed inside query or view modules.

- Initial message for **provider attribute deletion**.

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | DELETE_PROVIDER |
| Parameters | Attribute name (*AN*) |

1. Probing the input schema.

(a) Input schema searches for the attribute named as *AN* (denoted as *Q.I.A*).

(b) *Q.I.A* is deleted from input schema.

2. Probing the semantics schema.

   (a) Semantics schema searches if it has connection with the attribute *Q.I.A*.

   (b) If semantics schema has connection with *Q.I.A* then this sub tree of semantics is set to always true if the parent of the tree is an AND node or always false if the parent of the tree is an OR node.

3. Probing the output schema.

   (a) Output schema searches for the attributes that have connection with *Q.I.A* (denoted as *Q.O.As*).

   (b) If any *Q.O.A* exists, it is deleted from output schema.

- Initial message for **provider schema deletion**.

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | DELETE_PROVIDER |
| Parameters | Schema name (*SN*) |

1. Probing the input schema.

   (a) Input schema deletes itself.

2. Probing the semantics schema.

   (a) Semantics schema searches if it has connection with any of the attributes of input schema that got deleted (denoted as *Q.I.As*).

   (b) If semantics schema has connection with a *Q.I.A* then this sub tree of semantics is set to always true if the parent of the tree is an AND node or always false if the parent of the tree is an OR node.

3. Probing the output schema.

   (a) Output schema searches for the attributes that have connection with *Q.I.As* (denoted as *Q.O.As*).

   (b) If any *Q.O.A* exists, it is deleted from output schema.

- Initial message for **attribute deletion**.

| To node | Query/View |
|---|---|
| To schema | Query/View output schema |
| Event | DELETE_ATTRIBUTE |
| Parameters | Attribute name (*AN*) |

1. Probing the output schema.
   (a) Output schema searches for the attribute that is named as *AN* (denoted as *Q.O.A*).
   (b) *Q.O.A* is deleted.

- Initial message for **provider attribute renaming** (generated by a provider).

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | RENAME_PROVIDER |
| Parameters | Attribute name (*AN*) |

1. Probing the input schema.
   (a) Input schema searches for the attribute named as *AN* (denoted as *Q.I.A*).
   (b) *Q.I.A* is renamed with the name that was given to this rewrite maestro from the previous rewrite maestro.
2. Probing the output schema.
   (a) Output schema searches for the attributes that have connection with *Q.I.A* and are named as *AN* (denoted as (*Q.O.A*).
   (b) If *Q.O.A* exists, then it is renamed with the name that was given to this rewrite maestro from the previous rewrite maestro.

- Initial message for **provider schema renaming**.

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | RENAME_PROVIDER |
| Parameters | Schema name (*SN*) |

1. Probing the input schema.
   (a) Input schema is renamed with the name that was given to this rewrite maestro from the previous rewrite maestro.

- Initial message for **attribute renaming**.

| To node | Query/View |
|---|---|
| To schema | Query/View output schema |
| Event | RENAME_ATTRIBUTE |
| Parameters | Attribute name (*AN*) |

1. Probing the output schema.
   (a) Output schema searches for the attribute that is named as *AN* (denoted as *Q.O.A*).
   (b) User is asked for the new name of *Q.O.A* and then *Q.O.A* is renamed to what user selected if that name is not already present at the output schema of the module, then this name is propagated to other rewrite maestros.

- Initial message for **self deletion**.

| To node | Query/View |
|---|---|
| To schema | Query/View output schema |
| Event | DELETE_SELF |
| Parameters | |

1. Probing the input schema.
   (a) Input schema gets deleted.
2. Probing the semantics schema.
   (a) Semantics schema gets deleted.
3. Probing the output schema.
   (a) Output schema gets deleted.
   (b) Module node gets deleted.

- Initial message: **self renaming**.

| To node | Query/View |
|---|---|
| To schema | Query/View output schema |
| Event | RENAME_SELF |
| Parameters | |

1. Probing the input schemata.
   (a) User is asked for the new name of the module and then all input schemata are renamed to what user selected if that name is not already present at the graph, this name in the end is propagated to other rewrite maestros.
2. Probing the semantics schema.
   (a) Semantics schema is renamed to what user selected.
3. Probing the output schema.
   (a) Output schema is renamed to what user selected.

(b) Module node is rename to what user selected.

- Initial message for **provider attribute addition**.

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | ADD_ATTRIBUTE_PROVIDER |
| Parameters | |

1. Probing the input schema
   (a) Input schema adds a new attribute with the name given to this rewrite maestro from the previous rewrite maestro.
2. Probing the semantics schema
   (a) If semantics schema has a group by clause then user is asked if this new attribute should become a grouper or an aggregate function should be used for it.
3. Probing the output schema
   (a) Output schema adds a new attribute with the name given to this rewrite maestro from the previous rewrite maestro.

- Initial message for **attribute addition**.

| To node | Query/View |
|---|---|
| To schema | Query/View output schema |
| Event | ADD_ATTRIBUTE |
| Parameters | |

1. Probing the output schema.
   (a) User is prompted to select from its providers the attribute he wants to add to this module's output schema. If module has a group by clause in its semantics schema, user is asked if the new attribute should become a grouper or have an aggregate function over it. Then new attributes are added to output schema and to input provider schema (if attribute was not already there). Also the needed connections between nodes and semantics schema are made if they have to (for group by clause presence that was mentioned earlier).

- Incoming message for **alter semantics** (user generated).

| To node | Query/View |
|---|---|
| To schema | Query/View semantics schema |
| Event | ALTER_SEMANTICS |
| Parameters | |

1. Probing the semantics schema
   (a) User is prompted to write a new where and group by clause for the semantics schema. From the user input a new semantics tree is generated.

- Incoming message for **alter semantics** (system generated).

| To node | Query/View |
|---|---|
| To schema | Query/View input schema |
| Event | ALTER_SEMANTICS |
| Parameters | |

1. Here nothing happens, this is a system generated message that happened because user had asked for a change somewhere else (deletion of an attribute, addition of an attribute or alter of semantics of a module) and the spread of the messages to the graph, gave this message as result.

# Chapter 5

# Software architecture

Our project, Hecataeus, has its source code structured as described in Figure 5.1.

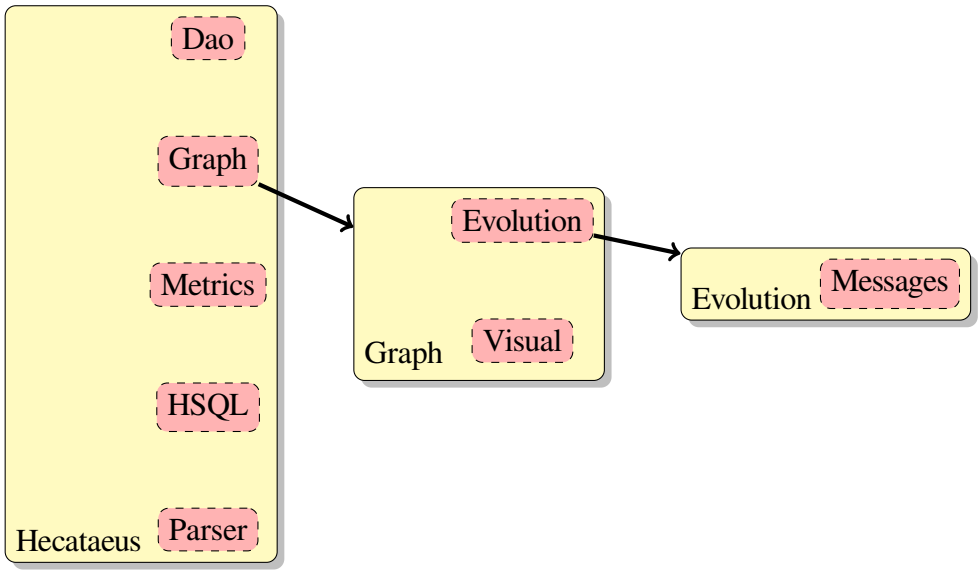



**Figure 5.1:** Hecataeus packages

The *Dao* package is responsible for the connection of Hecataeus to database management systems (DBMS). The main class in this package is HecataeusDatabase and it provides the needed functions for obtaining the table and view definitions of the database schema.

The *Graph* package is composed of i) the *Evolution* sub-package and ii) the *Visual* sub-package.

- The *Evolution* sub-package contains the $Evolution\ graph$ that describes the connections (edges of the architecture graph) between attributes, schemata and modules (nodes of the architecture graph). The Evolution graph is also responsible for the message propagation mechanism and rewriting methods contained in its sub-package Messages.
- The *Visual* sub-package contains the $Visual\ graph$, which extends the Evolution graph and is responsible for the representation of the evolution graph to users, by giving colors, shapes, icons and labels to the evolution nodes. The Visual graph is also responsible for the visibility of the nodes depending on the user selections (for example user may want to see the module graph or only the nodes that have status or all nodes contained in his graph). Finally, the visual graph is responsible for the topological layout of the nodes and the graph that user sees.

The $Metrics$ package is responsible for producing reports that have to do with the number of nodes of the graph, how many policies exist in the graph, the number of incoming and/or outgoing edges of nodes, the incoming and/or outgoing transitive (number of nodes that are connected to this one through the paths that this node is part of, as end of the path -for incoming- or as beginning of the path -for outgoing) degree of a node.

The $Hsql$ package is responsible for parsing the SQL statements from files containing a database schema and the views and queries that are related to that schema. In case of direct connections to databases through the $Dao$ package there is no need for such files, since Hecataeus can retrieve that information from a database connection.

The $Parser$ package is responsible for the creation of the graph by adding the needed nodes and edges, having as input the statements that Hsql previously parsed. The class that we have changed in order to implement the new model (with input, semantics and output schemata) was HecataeusGraphCreator. Besides the new model, a new "language" was created for the policies that were discussed in chapter 2.3, which was done on the HecataeusSQLExtensionParser class.

Finally, the $Hecataeus$ package is responsible for the communication of all the packages mentioned previously. Hecataeus package is also responsible for the graphic user interface that is needed for user to interact with the graph. For that reason we created the needed classes that provide the user the ability to interact with the policies of the graph and try an imminent change on the graph. We also added on Hecataeus package a class in order to organize as projects the graphs, policies and SQL files that a user works with.

## 5.1 Maestro implementation

As previously said, a *maestro* is a simple piece of software that is specialized on the combination $type\ of\ event \times module\ type$. Over the different design alternatives to be described next, we eventually decided to implement the maestros using the factory pattern for both the message propagation and rewrite mechanism, letting the future developers work flawlessly in the new event types that may arise. In case of a new event over a node or a new kind of nodes over the existing events, one should adapt the *create* function of MaestroFactory class, finally in the class that will handle the new combination one should implement the abstract functions *propagateMessages* and *doRewrite*. Alternatives over the selected implementation were the creation of a matrix containing as columns the $type\ of\ events$ and as rows the $module\ types$ and in the cells a generic algorithm that starts its execution in the node representing the module, and in status determination would: (1) ask modules schemata about the event (2) ask module about the event and set statuses according to the policy they have over the message and its parameters. This approach, although it has many advantages in metrics, would complicate the algorithms that we use in status determination. Another approach was to use a special design pattern where a class would instantiate all the available combinations of $type\ of\ event \times module\ type$ and when a new message would arrive, it would pass through all the available instances, where each instance would see if the message refers to it. This implementation would add additional complexity in the instances in order to check where a message refers to them or not.

As Figure 5.2 depicts, the class that implements the pattern is MaestroFactory which instantiates one of the following classes:

**QueryViewAddAttribute:** responsible for the addition of an attribute to the output schema of a Query or View module.

**QueryViewAddAttributeProvider:** responsible for the addition of an attribute to the output schema of a Query or View module, because one of the providers of the module added that attribute from its output schema.

**QueryViewAlterSemantics:** responsible for the change of the semantics tree.

**QueryViewDeleteAttribute:** responsible for the deletion of an attribute of a Query or View module, because one of the providers of the module deleted that attribute from its output schema.

**QueryViewDeleteProvider:** responsible for the deletion of an input schema and its attributes of a Query or View module, because one of the providers of the module was deleted.

**QueryViewDeleteSelf:** responsible for the deletion of a Query or View module.

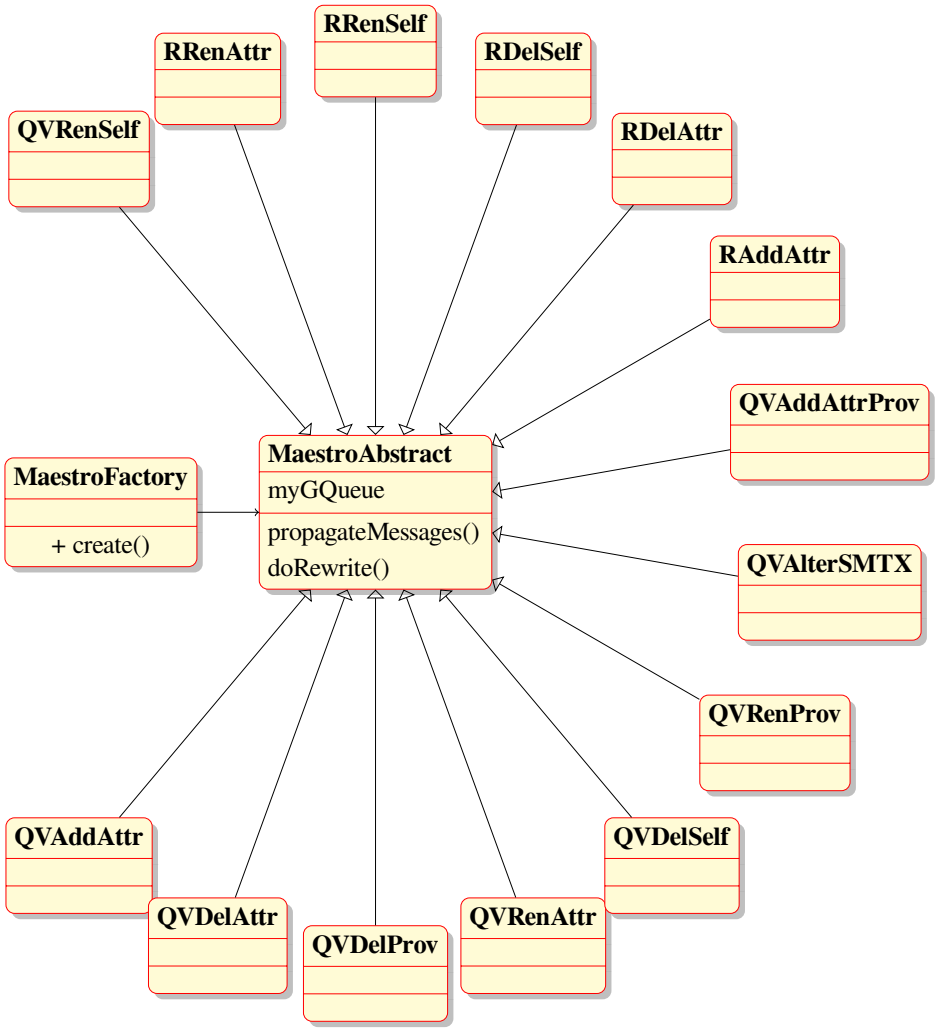



**Figure 5.2:** Factory method for maestros

**QueryViewRenameAttribute:** responsible for the renaming of an attribute of a Query or View module, because one of the providers of the module renamed that attribute to its output schema.

**QueryViewRenameProvider:** responsible for the renaming of an input schema of a Query or View module, because one of the providers of the module was renamed.

**QueryViewRenameSelf:** responsible for the renaming of a Query or View module.

**RelationAddAttribute:** responsible for the addition of an attribute to the output schema of a Relation module.

**RelationDeleteAttribute:** responsible for the deletion of an attribute from the output schema

of a module (regardless of module type).

**RelationDeleteSelf:** responsible for the deletion of a Relation module.

**RelationRenameAttribute:** responsible for the renaming of an attribute of an output schema of a module (regardless of the module type).

**RelationRenameSelf:** responsible for the renaming of a Relation module.

## 5.2 Our contribution

In the content of this thesis we have performed the following extensions to Hecataeus:

- We have implemented the rewriting mechanism letting the user of Hecataeus see how the ecosystem will look like after a change.
- The new model of architecture graph with input, semantics and output schemata, letting the user work over nicely isolated modules, with clear boundaries.
- The new message propagation mechanism, letting the user of Hecataeus confluently push events over the graph.
- New visualization to assign policies and handle events, clear from previously noisy graphical user interface of Hecataeus, using the observer pattern when the user selects a node for either changing its policies or initiating an event on it.
- We have implemented project management, letting the user of Hecataeus have an overview of scenarios at his disk.
- New language for policy annotation as described in Section 2.3.
- New Algorithms for policy checking, status determination and consumers notification have been implemented, as depicted in Figure 5.3, where:

  **FCASS:** searches in a schema node for the attribute that is named as message parameter and asks what policy it has for an event.

  **FCASSNO:** works as above and finally notifies the consumers of the module that are connected with the found attribute for the imminent change.

  **ASSS:** asks a schema node about its policy for an event.

**ASSSNO:** works as above and finally notifies the consumers of the module about the imminent change.

**AACSS:** asks all attributes of a schema node about their policy for an event.

**AACSSNO:** works as above and finally notifies the consumers of the module about the imminent change.

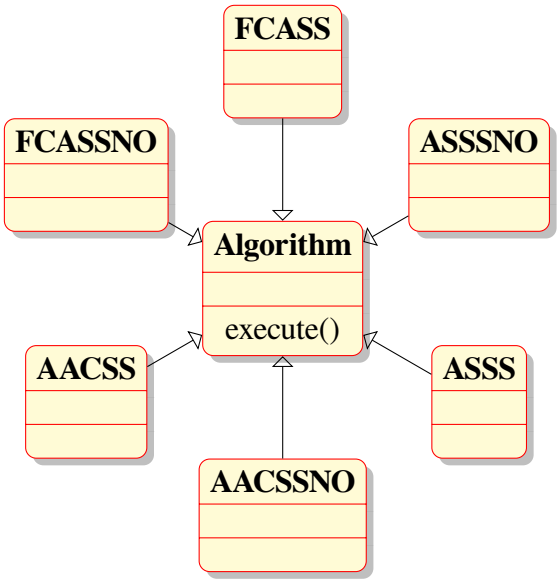



**Figure 5.3:** Check policy algorithms

# Chapter 6

# Experiments

We have performed several experiments in order to assess the behavior of our method. In the rest of this section, we discuss the setup of our experiments and then we move on to present our results, organized per research goal.

## 6.1 Experimental setup

In order to arrange our experimental setup, we need to fix the following parameters: (a) an ecosystem comprising a database schema surrounded by a set of queries and possibly a set of views, (b) a workload of events that are sequentially applied to the above configuration and (c) a palette of "profiles" that determine the way the ecosystem's architecture graph is annotated with policies towards the management of hypothetical events; hence, these profiles simulate the intention of the administrating team for the management of the ecosystem. We have employed the following ecosystems as the testbed for our experiments.

**University Ecosystem**. The first ecosystem that we have experimented with is the miniature ecosystem of the university database presented as reference example in the previous chapters.

**TPC-DS Ecosystem**. We have employed the TPC-DS schema as the testbed for our second experiment. TPC-DS is a benchmark that involves star schemata of a company that has the ability to $Sell$ and receive $Returns$ of its $Items$ with the following ways:(i) the $Web$ or (ii) a $Catalog$ or (iii) directly at the $Store$. Moreover, the company keeps data of $Customers$, regarding their $Income$ band, or their $Demographics$ data and additionally they keep data about the $Promotion$ of their $Items$. We used the TPC-DS version 1.1.0 [Cou12] which includes a set of database queries. Since our parser can not handle all kind of queries (e.g. queries containing keywords as LIMIT, HAVING etc) we had to remove parser offending parts

from any such queries in order to use them.

**Events**. Concerning the workload of events, we have created two workloads of events to test different contexts for the warehouse evolution. The distribution of events is shown in Table 6.1. The events were chosen randomly, having in mind to cover all the possible events that our implementation can handle. The first workload includes 48 events and is oriented towards changes of relation modules, whereas workload 2, that includes 14 events, is more oriented towards view modifications.

| **Operation** | **Workload 1** | **Workload 2** |
|---|---|---|
| Relation Delete | 2% | 13.3% |
| Relation Attribute Delete | 20.8% | 0% |
| Relation Rename | 8.3% | 13.3% |
| Relation Attribute Rename | 10.4% | 6.6% |
| Relation Add Attribute | 27.3% | 0% |
| Query/View Delete | 2% | 13.3% |
| Query/View Attribute Delete | 10.4% | 0% |
| Query/View Rename | 4.2% | 0% |
| Query/View Attribute Rename | 0% | 20.1% |
| Query/View Add Attribute | 14.6% | 13.3% |
| Query/View Alter Semantics | 0% | 20.1% |

**Table 6.1:** Two workloads of events for the TPC-DS ecosystem

**Policies**. We have annotated the graph with policies, in order to allow the management of evolution events. We have used two annotation "profiles", specifically: (a) *propagate all*, meaning that every change will be flooded to all the modules that should be notified about it and (b) *mixture*, consisting of 80% of the modules with propagate policies and 20% with blocking.

The first profile practically refers to a situation without any annotation. The second policy simulates a rather liberal environment, where most events are allowed to spread over the graph, so that their full impact can be observed; yet, 20% of modules are equipped with blocking policies to simulate the case of modules that should be handled with special care.

**Execution of experiments**. In our experiments, we have used an experimental prototype, Hecataeus, for the identification of the impact of hypothetical evolution events. For all the tested ecosystems, policy annotations and event testing, we have used the following sequence of actions. First, we annotate the architecture graph with policies. Next, we sequentially apply the events over the graph and in order to keep our measurements accurate we count their values each time. This means that each event is applied over the graph that resulted from the application of

the previous event. We also performed our experiments with hot cache (meaning that we did not keep the first output on all of our experiments) since the first event, consistently displayed 10 times bigger values in the time calculation measurement than the mean time of the rest events of the set[1]. For each event that we test, we monitor all the nodes of the graph. We collect measurements for the number of affected modules, along with the number of affected internal nodes, both for the phase of Status Determination and the phase of Rewriting. Moreover, we measure the elapsed time for each of the three phases: Status Determination, Path Check and Rewriting. All the experiments have been performed in a typical PC with the following setup: Intel Quad core CPU @ 2.66GHz with 1.9GB main memory.

## 6.2 Effectiveness of impact assessment and rewriting

In this experiment we evaluate the gain that we obtain for a developer that uses our tool.

The parameters of the experimentation are as follows. The architecture graph representing a database ecosystem with relations, views and queries. The policies used are all kinds of profiles that we have created. The workload of events is described in the first column of the following tables. For each policy, we vary the events and the nodes on which the events take place and in order to achieve our goal we made the following measurements that are displayed in the tables that follow:

[1] Although we tried to perform our experiments with hot cache, there still might be a small overhead on the execution of some events

| Annotation | Meaning |
|---|---|
| $AM$ | Number of affected modules: for each event, we measure the number of modules with status either BLOCK or PROPAGATE. We exclude the object that initiates the sequence of events from the computation, as it would be counted in both occasions. |
| $\%AM$ | Effort gains of the developer at module level. We assess the gain of a developer using Hecataeus compared to the situation where he would have to perform all checks by hand. This gains amounts to the percentage of useless checks the user would have made. We exclude the object that initiates the sequence of events from the computation, as it would be counted in both occasions. Formally, $\%AM$ is given by the equation 6.1. |
| $AI$ | Number of affected internal nodes: for each event, we measure the number of the internal nodes of the graph with status either BLOCK or PROPAGATE. We exclude the object that initiates the sequence of events from the computation, as it would be counted in both occasions. |
| $\%AI$ | Effort gains of the developer at node level. We assess the gain of a developer using Hecataeus compared to the situation where he would have to perform all checks by hand. This gains amounts to the percentage of useless checks the user would have made. We exclude the object that initiates the sequence of events from the computation, as it would be counted in both occasions. Formally, $\%AI$ is given by the equation 6.2. |
| $NM$ | Number of cloned modules: for each event, we measure the number of modules that were cloned. |
| $ERM$ | Number of existing adapted modules: for each event, we measure the number of modules that adapted the change |
| $RM$ | Sum of cloned and existing adapted modules |

**Table 6.2:** Annotations of column names and meaning

The formula for the definition of developer gains are the following:

$$\%AM = 1 - \frac{\#Affected\ Modules}{\#(Queries \cup Views)} \tag{6.1}$$

$$\%AI = 1 - \frac{\#Affected\ Internal\ nodes}{\#Internal\ nodes\ of\ (Queries \cup Views)} \tag{6.2}$$

In the following tables, we will refer to the type of event performed via the abbreviations of Table 6.3.

| Abbreviation | Meaning |
|---|---|
| RS | Rename self |
| DS | Delete self |
| AA | Add attribute |
| AS | Alter semantics |

**Table 6.3:** Abbreviations of events

### 6.2.1 University ecosystem

**Propagate policy**. In this experiment, we annotate the University ecosystem with *propagate* policy for all its nodes and apply a small workload of events described in detail in Table 6.4.

| | Impact assessment | | | | Adaptation assessment | | |
|---|---|---|---|---|---|---|---|
| **Event:Node** | **AM** | **% AM** | **AI** | **% AI** | **NM** | **ERM** | **RM** |
| RS:V_COURSE | 2 | **50** | 2 | 98.37 | 0 | 2 | **2** |
| RS:V_TR | 3 | **25** | 4 | 96.75 | 0 | 3 | **3** |
| RS:SEMESTER.MDESCR | 2 | **50** | 10 | 91.87 | 0 | 3 | **3** |
| RS:TRANSCRIPT.SID | 3 | **25** | 14 | 88.62 | 0 | 4 | **4** |
| RS:TRANSCRIPT.TGRADE | 3 | **25** | 12 | 90.24 | 0 | 4 | **4** |
| RS:TRANSCRIPT | 1 | **75** | 2 | 98.37 | 0 | 2 | **2** |
| AA:TRANSCRIPT | 3 | **25** | 11 | 91.06 | 0 | 4 | **4** |
| AA:STUDENT | 1 | **75** | 4 | 96.97 | 0 | 2 | **2** |
| AA:COURSESTD | 4 | **0** | 14 | 89.63 | 0 | 5 | **5** |
| AA:SEMESTER | 4 | **0** | 14 | 90.41 | 0 | 5 | **5** |
| AA:V_COURSE | 4 | **0** | 11 | 92.99 | 0 | 4 | **4** |
| DS:SEMESTER.SEMDESCR | 2 | **50** | 10 | 94.01 | 0 | 3 | **3** |
| DS:TRANSCRIPT.CID | 3 | **25** | 8 | 95.06 | 0 | 4 | **4** |
| DS:COURSESTD | 4 | **0** | 54 | 66.25 | 0 | 5 | **5** |
| Minimum | 1 | **0** | 2 | 66.25 | 0 | 2 | **2** |
| Maximum | 4 | **75** | 54 | 98.37 | 0 | 5 | **5** |
| Average | 2.79 | **30.36** | 12.14 | 91.47 | 0 | 3.57 | **3.57** |

**Table 6.4:** University database - User benefit for the propagate all profile

As someone can see in Table 6.4, when the *propagate all* policy is used, the user gains on average 30% in module checking. This happens because our reference example ecosystem is too cohesive, with both queries depending only in one view for their input, meaning that a change in a relation that is provider of this view will have to check all the queries and views of the graph.

**Mixture 20% block - 80% propagate policy**. In this experiment, we annotate the University ecosystem with *mixture* policy for its nodes and apply a small workload of events described in detail in Table 6.5.

The blocking modules on our reference example are modules Q2 and STUDENT, depicted in Figure 6.1 with red color, painted in thicker line, placed in the upper right corner of the figure.

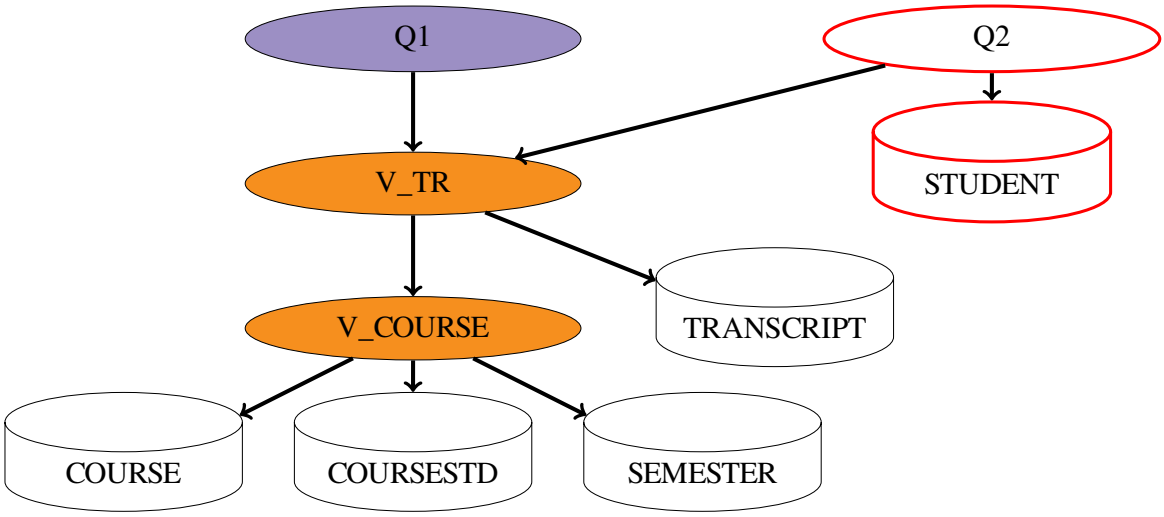



**Figure 6.1:** Reference Example - Mixture policies

| | Impact assessment | | | | Adaptation assessment | | |
|---|---|---|---|---|---|---|---|
| **Event:Node** | **AM** | **% AM** | **AI** | **% AI** | **NM** | **ERM** | **RM** |
| RS:V_COURSE | 2 | **50** | 2 | 98.37 | 0 | 2 | **2** |
| RS:V_TR | 3 | **25** | 4 | 96.75 | 1 | 2 | **3** |
| RS:SEMESTER.MDESCR | 3 | **40** | 14 | 90.28 | 0 | 4 | **4** |
| RS:TRANSCRIPT.SID | 4 | **20** | 18 | 87.5 | 0 | 0 | **0** |
| RS:TRANSCRIPT.TGRADE | 4 | **20** | 16 | 88.89 | 0 | 0 | **0** |
| RS:TRANSCRIPT | 2 | **60** | 3 | 97.92 | 0 | 3 | **3** |
| AA:TRANSCRIPT | 4 | **20** | 14 | 90.28 | 0 | 0 | **0** |
| AA:STUDENT | 0 | **100** | 1 | 99.31 | 0 | 0 | **0** |
| AA:COURSESTD | 5 | **0** | 17 | 88.19 | 0 | 0 | **0** |
| AA:SEMESTER | 5 | **0** | 17 | 88.19 | 0 | 0 | **0** |
| AA:V_COURSE | 5 | **0** | 14 | 90.28 | 2 | 5 | **7** |
| DS:SEMESTER.SEMDESCR | 5 | **28.57** | 22 | 89 | 0 | 7 | **7** |
| DS:TRANSCRIPT.CID | 5 | **28.57** | 16 | 91.53 | 0 | 0 | **0** |
| DS:COURSESTD | 7 | **0** | 64 | 66.14 | 0 | 0 | **0** |
| Minimum | 0 | **0** | 1 | 66.14 | 0 | 0 | **0** |
| Maximum | 7 | **100** | 64 | 99.31 | 2 | 7 | **7** |
| Average | 3.86 | **28.01** | 15.86 | 90.19 | 0.21 | 1.64 | **1.86** |

**Table 6.5:** University database - User benefit for the propagate all profile

As previously discussed in *propagate all* policy, since our reference example is too cohesive and the fact that not all modules propagate the events (which means that some of the views might be cloned on a blocking event) makes the average profit of the user a little smaller (28%) in module checking.

### 6.2.2 TPC-DS ecosystem

We now move on to assess our method on a more realistic environment, the TPC-DS ecosystem. TPC-DS consists of 15 relations, 5 views and 27 queries. We examine TPC-DS with two policy annotation profiles (propagate all and mixture) and two workloads of events (see Table 6.1).

**Workload 1 - Propagate policy**. In this experiment, we annotate the TPC-DS ecosystem with *propagate* policy for all its nodes and apply a workload, as described in detail in Table 6.6.

| | Impact assessment | | | | Adaptation assessment | | |
|---|---|---|---|---|---|---|---|
| **Event:Node** | **AM** | **% AM** | **AI** | **% AI** | **NM** | **ERM** | **RM** |
| DS:WEB_SALES | 0 | **100** | 6 | 99.59 | 0 | 1 | **1** |
| DS:INCOME_BAND.IB_INCOME_BAND_SK | 2 | **93.75** | 10 | 99.32 | 0 | 3 | **3** |
| DS:Q25 | 1 | **96.88** | 4 | 99.73 | 0 | 1 | **1** |
| DS:CUSTOMER_ADDRESS.CA_STREET_NAME | 0 | **100** | 2 | 99.86 | 0 | 1 | **1** |
| RS:STRORE_RETURNS | 3 | **90.32** | 4 | 99.72 | 0 | 4 | **4** |
| RS:CUSTOMER_DEMOGRAPHICS | 3 | **90.32** | 4 | 99.72 | 0 | 4 | **4** |
| RS:HOUSEHOLD_DEMOGRGRAPHICS | 3 | **90.32** | 4 | 99.72 | 0 | 4 | **4** |
| RS:VIEW38 | 2 | **93.55** | 2 | 99.86 | 0 | 2 | **2** |
| RS:CUSTOMER_TOTAL_RET | 3 | **90.32** | 3 | 99.79 | 0 | 3 | **3** |
| RS:INVENTORY | 1 | **96.77** | 2 | 99.86 | 0 | 2 | **2** |
| RS:INVENTORY.INV_WAREHOUSE_SK | 0 | **100** | 2 | 99.86 | 0 | 1 | **1** |
| DS:INVENTORY.INV_WAREHOUSE_SK | 0 | **100** | 2 | 99.86 | 0 | 1 | **1** |
| DS:ITEM.I_ITEM_SK | 9 | **70.97** | 39 | 97.27 | 0 | 10 | **10** |
| RS:CUSTOMER_ADDRESS.CA_ADDRESS_SK | 8 | **74.19** | 18 | 98.74 | 0 | 9 | **9** |
| RS:HOUSEHOLD_DEMOGRGRAPHICS.HD_DEMO_SK | 3 | **90.32** | 8 | 99.44 | 0 | 4 | **4** |
| RS:CATALOG_SALES.CS_ORDER_NUMBER | 0 | **100** | 2 | 99.86 | 0 | 1 | **1** |
| RS:CUSTOMER.C_CUSTOMER_ID | 6 | **80.65** | 18 | 98.74 | 0 | 7 | **7** |
| AA:INVENTORY | 1 | **96.77** | 4 | 99.72 | 0 | 2 | **2** |
| AA:CUSTOMER_DEMOGRAPHICS | 3 | **90.32** | 10 | 99.3 | 0 | 4 | **4** |
| AA:DATE_DIM | 24 | **22.58** | 80 | 94.44 | 0 | 25 | **25** |
| AA:CUSTOMER | 15 | **51.61** | 49 | 96.74 | 0 | 16 | **16** |
| AA:STORE_SALES | 14 | **54.84** | 46 | 97.01 | 0 | 15 | **15** |
| AA:ITEM | 9 | **70.97** | 28 | 98.22 | 0 | 10 | **10** |
| AA:SSTORE | 7 | **77.42** | 22 | 98.62 | 0 | 8 | **8** |
| AA:INCOME_BAND | 2 | **93.55** | 7 | 99.56 | 0 | 3 | **3** |
| AA:CUSTOMER_ADDRESS | 10 | **67.74** | 32 | 98.01 | 0 | 11 | **11** |
| AA:HOUSEHOLD_DEMOGRGRAPHICS | 4 | **87.1** | 13 | 99.2 | 0 | 5 | **5** |
| AA:PROMOTION | 2 | **93.55** | 7 | 99.57 | 0 | 3 | **3** |
| AA:CATALOG_SALES | 5 | **83.87** | 16 | 99.03 | 0 | 6 | **6** |
| AA:STRORE_RETURNS | 5 | **83.87** | 16 | 99.04 | 0 | 6 | **6** |
| AA:VIEW38 | 2 | **93.55** | 4 | 99.76 | 0 | 2 | **2** |
| AA:CUSTOMER_TOTAL_RETRN | 3 | **90.32** | 7 | 99.58 | 0 | 3 | **3** |
| AA:Q17 | 1 | **96.77** | 1 | 99.94 | 0 | 1 | **1** |
| AA:Q4 | 1 | **96.77** | 1 | 99.94 | 0 | 1 | **1** |
| AA:Q3 | 1 | **96.77** | 1 | 99.94 | 0 | 1 | **1** |
| AA:Q6 | 1 | **96.77** | 1 | 99.94 | 0 | 1 | **1** |
| AA:Q27 | 1 | **96.77** | 1 | 99.94 | 0 | 1 | **1** |
| DS:ITEM.I_CONTAINER | 0 | **100** | 2 | 99.88 | 0 | 1 | **1** |
| DS:WEB_RETURNS.WR_RETURNED_DATE_SK | 3 | **90.32** | 8 | 99.53 | 0 | 4 | **4** |
| DS:CUSTOMER_DEMOGRAPHICS.CD_MARITAL_STATUS | 2 | **93.55** | 10 | 99.41 | 0 | 3 | **3** |
| DS:Q26.CA_ZIP | 1 | **96.77** | 2 | 99.88 | 0 | 1 | **1** |
| DS:Q12.STORE_RETURNS_LOSS | 1 | **96.77** | 2 | 99.88 | 0 | 1 | **1** |
| DS:STRORE_RETURNS.SR_RETURN_TAX | 0 | **100** | 2 | 99.88 | 0 | 1 | **1** |
| DS:CUSTOMER.C_CUSTOMER_SK | 11 | **64.52** | 38 | 97.74 | 0 | 12 | **12** |
| DS:CUSTOMER.C_CUSTOMER_ID | 6 | **80.65** | 38 | 97.72 | 0 | 7 | **7** |
| DS:CUSTOMER_TOTAL_RET.CTR_STORE_SK | 3 | **90.32** | 10 | 99.4 | 0 | 3 | **3** |
| DS:VIEW38.D_DATE | 2 | **93.55** | 4 | 99.76 | 0 | 2 | **2** |
| DS:CUSTOMER_TOTAL_RETRN.CTR_STATE | 2 | **93.55** | 6 | 99.64 | 0 | 2 | **2** |
| Minimum | 0 | **22.58** | 1 | 94.44 | 0 | 1 | **1** |
| Maximum | 24 | **100** | 80 | 99.94 | 0 | 25 | **25** |
| Average | 3.88 | **87.51** | 12.46 | 99.19 | 0 | 4.56 | **4.56** |

**Table 6.6:** TPC-DS database - Workload 1 - User benefit for the propagate all profile

In the case of the TPC-DS ecosystem we can see that the user gains on average 88% in module checking, having the smallest gain when the events took place on relations with high in-degree (meaning that these relations are used by many queries/views), as for example Date_Dim. This means that if user was not equipped with Hecataeus, he would have to perform useless checks on more than 98% of the ecosystem's queries in module level for the 48 events. The numbers go very high when the internals are considered; however this is due also to the fact that we had only a few module deletions that would affect many internal nodes on the consumers of the modules.

In term of rewritings, the system reports 4.5 module rewritings on average, with a 10% of occasions with more than 10 modules containing rewritings. Observe that due to the propagate all nature of the profile, there are no clones produced.

**Workload 1 - Mixture 20% block - 80% propagate policy**. In this experiment, we annotate the TPC-DS ecosystem with *mixture* policy for its nodes and apply a workload, as described in detail in Table 6.7. The nodes that were assigned the BLOCK policy for all the incoming messages were: (i) WEB_RETURNS, (ii) INCOME_BAND, (iii) Q1, (iv) Q3, (v) Q4, (vi) Q17, (vii) Q19, (viii) Q20 and (ix) Q21.

| | Impact assessment | | | | Adaptation assessment | | |
|---|---|---|---|---|---|---|---|
| **Event:Node** | **AM** | **% AM** | **AI** | **% AI** | **NM** | **ERM** | **RM** |
| DS:WEB_SALES | 0 | **100** | 6 | 99.59 | 0 | 1 | **1** |
| DS:INCOME_BAND.IB_INCOME_BAND_SK | 0 | **100** | 2 | 99.86 | 0 | 0 | **0** |
| DS:Q25 | 1 | **96.88** | 4 | 99.73 | 0 | 1 | **1** |
| DS:CUSTOMER_ADDRESS.CA_STREET_NAME | 0 | **100** | 2 | 99.86 | 0 | 1 | **1** |
| RS:STRORE_RETURNS | 3 | **90.32** | 4 | 99.72 | 0 | 4 | **4** |
| RS:CUSTOMER_DEMOGRAPHICS | 3 | **90.32** | 4 | 99.72 | 0 | 4 | **4** |
| RS:HOUSEHOLD_DEMOGRGRAPHICS | 3 | **90.32** | 4 | 99.72 | 0 | 4 | **4** |
| RS:VIEW38 | 2 | **93.55** | 2 | 99.86 | 1 | 1 | **2** |
| RS:CUSTOMER_TOTAL_RET | 3 | **90.63** | 3 | 99.79 | 1 | 2 | **3** |
| RS:INVENTORY | 1 | **96.97** | 2 | 99.86 | 0 | 0 | **0** |
| RS:INVENTORY.INV_WAREHOUSE_SK | 0 | **100** | 2 | 99.86 | 0 | 1 | **1** |
| DS:INVENTORY.INV_WAREHOUSE_SK | 0 | **100** | 2 | 99.86 | 0 | 1 | **1** |
| DS:ITEM.I_ITEM_SK | 9 | **72.73** | 39 | 97.36 | 0 | 0 | **0** |
| RS:CUSTOMER_ADDRESS.CA_ADDRESS_SK | 8 | **75.76** | 18 | 98.78 | 0 | 0 | **0** |
| RS:HOUSEHOLD_DEMOGRGRAPHICS.HD_DEMO_SK | 3 | **90.91** | 8 | 99.46 | 0 | 4 | **4** |
| RS:CATALOG_SALES.CS_ORDER_NUMBER | 0 | **100** | 2 | 99.86 | 0 | 1 | **1** |
| RS:CUSTOMER.C_CUSTOMER_ID | 6 | **81.82** | 18 | 98.78 | 0 | 0 | **0** |
| AA:INVENTORY | 1 | **96.97** | 4 | 99.73 | 0 | 0 | **0** |
| AA:CUSTOMER_DEMOGRAPHICS | 3 | **90.91** | 10 | 99.32 | 0 | 4 | **4** |
| AA:DATE_DIM | 26 | **21.21** | 86 | 94.2 | 0 | 0 | **0** |
| AA:CUSTOMER | 16 | **51.52** | 52 | 96.49 | 0 | 0 | **0** |
| AA:STORE_SALES | 15 | **54.55** | 49 | 96.69 | 0 | 0 | **0** |
| AA:ITEM | 9 | **72.73** | 28 | 98.11 | 0 | 0 | **0** |
| AA:SSTORE | 7 | **78.79** | 22 | 98.52 | 0 | 0 | **0** |
| AA:INCOME_BAND | 0 | **100** | 1 | 99.93 | 0 | 0 | **0** |
| AA:CUSTOMER_ADDRESS | 10 | **69.7** | 32 | 97.84 | 0 | 0 | **0** |
| AA:HOUSEHOLD_DEMOGRGRAPHICS | 4 | **87.88** | 13 | 99.12 | 0 | 0 | **0** |
| AA:PROMOTION | 2 | **93.94** | 7 | 99.53 | 0 | 3 | **3** |
| AA:CATALOG_SALES | 6 | **81.82** | 19 | 98.72 | 0 | 0 | **0** |
| AA:STRORE_RETURNS | 6 | **81.82** | 19 | 98.72 | 0 | 0 | **0** |
| AA:VIEW38 | 2 | **93.94** | 4 | 99.73 | 1 | 1 | **2** |
| AA:CUSTOMER_TOTAL_RETRN | 3 | **91.18** | 7 | 99.54 | 1 | 2 | **3** |
| AA:Q17 | 1 | **97.14** | 1 | 99.94 | 0 | 0 | **0** |
| AA:Q4 | 1 | **97.14** | 1 | 99.94 | 0 | 0 | **0** |
| AA:Q3 | 1 | **97.14** | 1 | 99.94 | 0 | 0 | **0** |
| AA:Q6 | 1 | **97.14** | 1 | 99.94 | 0 | 1 | **1** |
| AA:Q27 | 1 | **97.14** | 1 | 99.94 | 0 | 1 | **1** |
| DS:ITEM.I_CONTAINER | 0 | **100** | 2 | 99.87 | 0 | 1 | **1** |
| DS:WEB_RETURNS.WR_RETURNED_DATE_SK | 0 | **100** | 2 | 99.87 | 0 | 0 | **0** |
| DS:CUSTOMER_DEMOGRAPHICS.CD_MARITAL_STATUS | 2 | **94.29** | 10 | 99.35 | 0 | 3 | **3** |
| DS:Q26.CA_ZIP | 1 | **97.14** | 2 | 99.87 | 0 | 1 | **1** |
| DS:Q12.STORE_RETURNS_LOSS | 1 | **97.14** | 2 | 99.87 | 0 | 1 | **1** |
| DS:STRORE_RETURNS.SR_RETURN_TAX | 0 | **100** | 2 | 99.87 | 0 | 1 | **1** |
| DS:CUSTOMER.C_CUSTOMER_SK | 13 | **62.86** | 46 | 97.01 | 0 | 0 | **0** |
| DS:CUSTOMER.C_CUSTOMER_ID | 7 | **80** | 40 | 97.4 | 0 | 0 | **0** |
| DS:CUSTOMER_TOTAL_RET.CTR_STORE_SK | 3 | **91.43** | 10 | 99.35 | 1 | 2 | **3** |
| DS:VIEW38.D_DATE | 2 | **94.44** | 4 | 99.74 | 1 | 1 | **2** |
| DS:CUSTOMER_TOTAL_RETRN.CTR_STATE | 2 | **94.59** | 6 | 99.62 | 0 | 2 | **2** |
| Minimum | 0 | **21.21** | 1 | 94.2 | 0 | 0 | **0** |
| Maximum | 26 | **100** | 86 | 99.94 | 1 | 4 | **4** |
| Average | 3.92 | **88.22** | 12.63 | 99.15 | 0.13 | 1.02 | **1.15** |

**Table 6.7:** TPC-DS database - Workload 1 - User benefit for the mixture profile

We observe a large drop in the number of rewritten modules. This is due to the blocking nature of the annotation of a subset of the graph. Most of our blocker modules are queries that use a great number of relations (the only relations that do not have connection with any of the blocker queries are PROMOTION and CUSTOMER_DEMOGRAPHICS). Since our workload contains many events happening on relations and our algorithm does not perform any rewriting when there is a veto of event that started from a relation node, this is something we were expecting. The effort gains of the developer in module checking, when the *mixture* policy is used, is the same as the gain of *propagate all* policy. The minimum gain is also smaller here (21% as opposed to 23%) because of the cloning of some views that happened in the previous events, which got connected to the Date_Dim relation.

**Workload 2 - Propagate policy**. We now move on to assess our method on TPC-DS using a new workload that is more oriented towards view modifications. In this experiment, we annotate the TPC-DS ecosystem with *propagate* policy for all its nodes and apply workload 2, as described in detail in Table 6.9.

| | Impact assessment | | | | Adaptation assessment | | |
|---|---|---|---|---|---|---|---|
| **Event:Node** | **AM** | **% AM** | **AI** | **% AI** | **NM** | **ERM** | **RM** |
| DS:WEB_SALES | 0 | **100** | 6 | 99.59 | 0 | 1 | **1** |
| RS:CUSTOMER_DEMOGRAPHICS.CD_DEMO_SK | 3 | **90.63** | 8 | 99.46 | 0 | 4 | **4** |
| RS:VIEW38.C_LAST_NAME | 2 | **93.75** | 4 | 99.73 | 0 | 2 | **2** |
| RS:CUSTOMER_TOTAL_RET.CTR_TOTAL_RETURN | 2 | **93.75** | 4 | 99.73 | 0 | 2 | **2** |
| RS:CUSTOMER_TOTAL_RETRN.CTR_TOTAL_RETURN | 2 | **93.75** | 6 | 99.59 | 0 | 2 | **2** |
| AS:VIEW38 | 2 | **93.75** | 2 | 99.86 | 0 | 2 | **2** |
| AS:CUSTOMER_TOTAL_RET | 3 | **90.63** | 3 | 99.8 | 0 | 3 | **3** |
| AS:CUSTOMER_TOTAL_RETRN | 3 | **90.63** | 3 | 99.8 | 0 | 3 | **3** |
| AA:VIEW38 | 2 | **93.75** | 4 | 99.73 | 0 | 2 | **2** |
| AA:Q18 | 1 | **96.88** | 1 | 99.93 | 0 | 1 | **1** |
| DS:Q18 | 1 | **96.88** | 3 | 99.8 | 0 | 1 | **1** |
| DS:CUSTOMER_DEMOGRAPHICS | 3 | **90.32** | 34 | 97.68 | 0 | 4 | **4** |
| RS:ITEM | 10 | **67.74** | 11 | 99.24 | 0 | 11 | **11** |
| RS:PROMOTION | 2 | **93.55** | 3 | 99.79 | 0 | 3 | **3** |
| Minimum | 0 | **67.74** | 1 | 97.68 | 0 | 1 | **1** |
| Maximum | 10 | **100** | 34 | 99.93 | 0 | 11 | **11** |
| Average | 2.57 | **91.86** | 6.57 | 99.55 | 0 | 2.93 | **2.93** |

**Table 6.8:** TPC-DS database - Workload 2 - User benefit for the propagate all profile

In the case of workload 2, the average gain in module checking for the user is 92%. Comparing the minimum gain in both workloads we observe that it is now 68%. This happens because, on workload 2 we didnot use the Date_Dim relation (the relation with the highest in-degree in our ecosystem) in any of our events. Actually, the module with the highest in-degree used in workload 2 is relation Item, which has nearly half incoming edges compared to Date_Dim. Observe equation 6.1: the numerator of our fraction is smaller (due to smaller in-degree) and thus the fraction is smaller, resulting in higher benefits. We also observe an increase of the

gain. This happens because the workload has only three events related to relations that would affect a majority of modules.

**Workload 2 - Mixture 20% block - 80% propagate policy**. As in Workload 1, the nodes that were assigned the BLOCK policy for all the incoming messages were: (i) WEB_RETURNS, (ii) INCOME_BAND, (iii) Q1, (iv) Q3, (v) Q4, (vi) Q17, (vii) Q19, (viii) Q20 and (ix) Q21 out of a total of 47 modules.

| | Impact assessment | | | | Adaptation assessment | | |
|---|---|---|---|---|---|---|---|
| **Event:Node** | **AM** | **% AM** | **AI** | **% AI** | **NM** | **ERM** | **RM** |
| DS:WEB_SALES | 0 | **100** | 6 | 99.59 | 0 | 1 | **1** |
| RS:CUSTOMER_DEMOGRAPHICS.CD_DEMO_SK | 3 | **90.63** | 8 | 99.46 | 0 | 4 | **4** |
| RS:VIEW38.C_LAST_NAME | 2 | **93.75** | 4 | 99.73 | 1 | 1 | **2** |
| RS:CUSTOMER_TOTAL_RET.CTR_TOTAL_RETURN | 2 | **93.94** | 4 | 99.73 | 1 | 1 | **2** |
| RS:CUSTOMER_TOTAL_RETRN.CTR_TOTAL_RETURN | 2 | **94.12** | 6 | 99.6 | 1 | 1 | **2** |
| AS:VIEW38 | 2 | **94.29** | 2 | 99.87 | 1 | 1 | **2** |
| AS:CUSTOMER_TOTAL_RET | 3 | **91.67** | 3 | 99.81 | 1 | 2 | **3** |
| AS:CUSTOMER_TOTAL_RETRN | 3 | **91.89** | 3 | 99.81 | 1 | 2 | **3** |
| AA:VIEW38 | 2 | **94.74** | 4 | 99.75 | 1 | 1 | **2** |
| AA:Q18 | 1 | **97.44** | 1 | 99.94 | 0 | 1 | **1** |
| DS:Q18 | 1 | **97.44** | 3 | 99.82 | 0 | 1 | **1** |
| DS:CUSTOMER_DEMOGRAPHICS | 3 | **92.11** | 34 | 97.9 | 0 | 4 | **4** |
| RS:ITEM | 10 | **73.68** | 11 | 99.32 | 0 | 0 | **0** |
| RS:PROMOTION | 2 | **94.74** | 3 | 99.81 | 0 | 3 | **3** |
| Minimum | 0 | **73.68** | 1 | 97.9 | 0 | 0 | **0** |
| Maximum | 10 | **100** | 34 | 99.94 | 1 | 4 | **4** |
| Average | 2.57 | **92.89** | 6.57 | 99.58 | 0.5 | 1.64 | **2.14** |

**Table 6.9:** TPC-DS database - Workload 2 - User benefit for the mixture profile

Here we can see more clearly that modules get cloned. This fact, is the reason that the minimum gain of the user is increasing (causing the average gain to also increase). Since we create new modules that are not connected to relation Item we actually increase the denominator of the fraction used in our gain equation 6.1

## 6.3 Efficiency

In this subsection, we study the behavior of the three algorithms needed to perform impact assessment and adaptation with respect to the time needed for their completion. We will focus the discussion on the first workload of events for TPC-DS and complement it with the deviations observed in the rest of the experiments that we have performed.

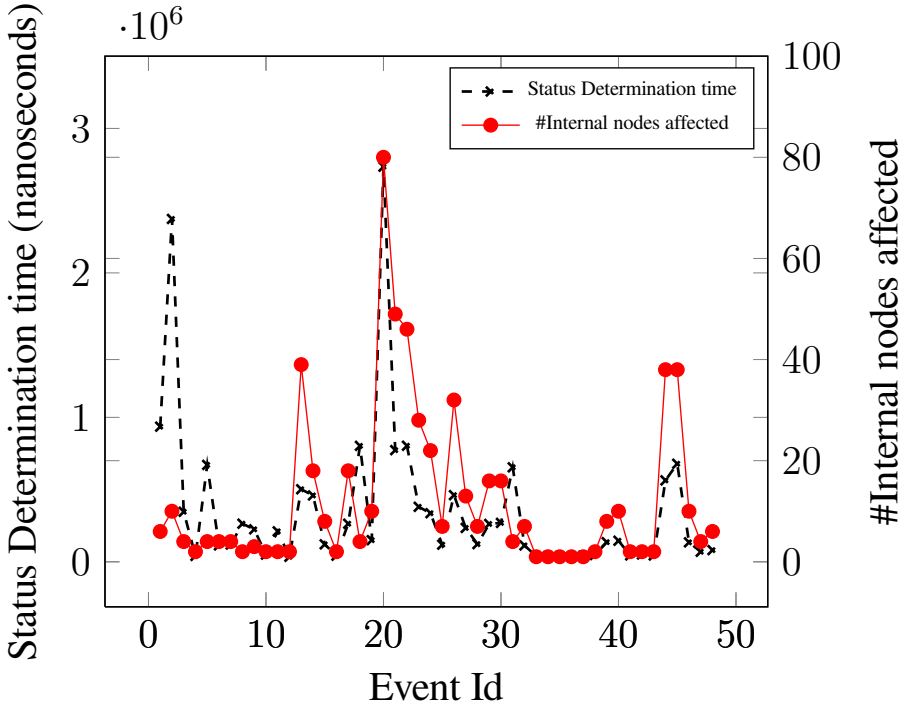
**Time breakdown**. As one can see, the average time needed for the status determination is quite similar in both cases of *propagate all* and *mixture* profiles. In the case of *mixture*, the average status determination time and the average rewriting time are slightly lower due to the

blockers (i.e., in some cases, there is no propagation of the events). For exactly the same reason, however, the path check time is more 3 times higher in the *mixture* case: whenever blockers exist, we possibly need to retain more than one variants per view, thus resulting in higher path check times; on the contrary, *propagate all* has no blockers and no variants. The percentage of time spent in the different tasks is equally divided between status determination and rewriting with a very small fraction of time to path check.

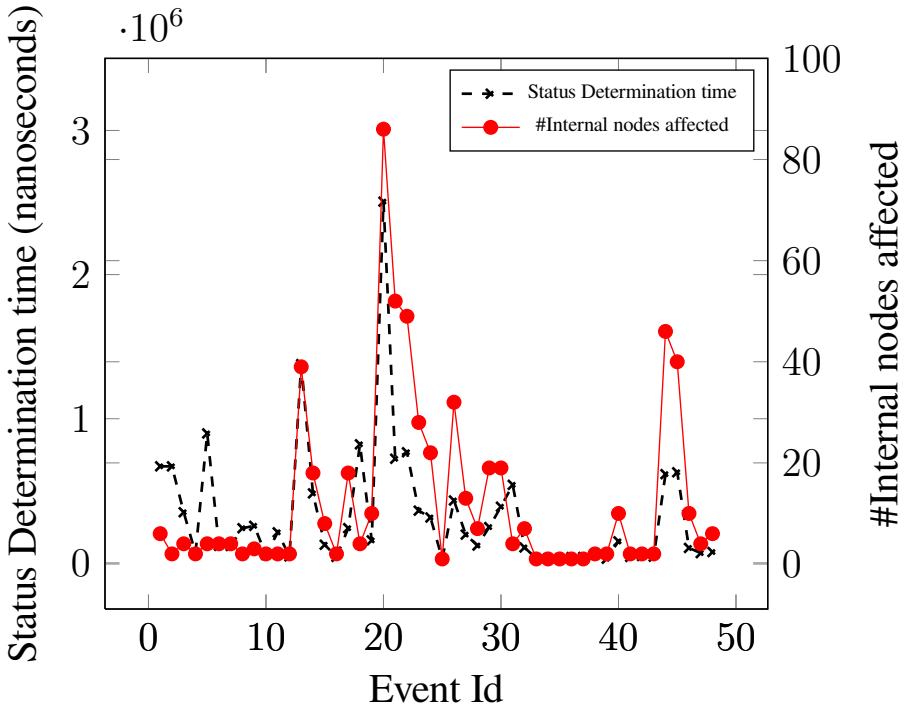
| | Average time (nanosecs) | | | | Percentage Breakdown | | |
|---|---|---|---|---|---|---|---|
| | Status Determination | Path Check | Rewriting | Total | Status Determination | Path Check | Rewriting |
| Propagate all | 358161 | 4947 | 367071 | 730179 | 49% | 1% | 50% |
| Mixture | 327488 | 18340 | 341735 | 687563 | 48% | 2% | 50% |

**Time spent vs number of modules affected**. What is the relationship of time spent versus the number of modules affected by a potential event?

We assess the time spent for status determination in each event and compare it to the number of internal nodes affected by the event. By *internal nodes*, we refer to the nodes that are internal to the modules of the graph (e.g., schema attributes, comparison nodes in selection conditions, etc). In Figure 6.2, the horizontal axis depicts the sequence of events. The left vertical axis depicts the time spent for status determination (with the respective data depicted via a dotted blue line) and the right vertical axis depicts the number of affected internal nodes (with the respective data depicted via a solid red line). We employ lines instead of scatter plots only for intuition purposes.




(a) Propagate



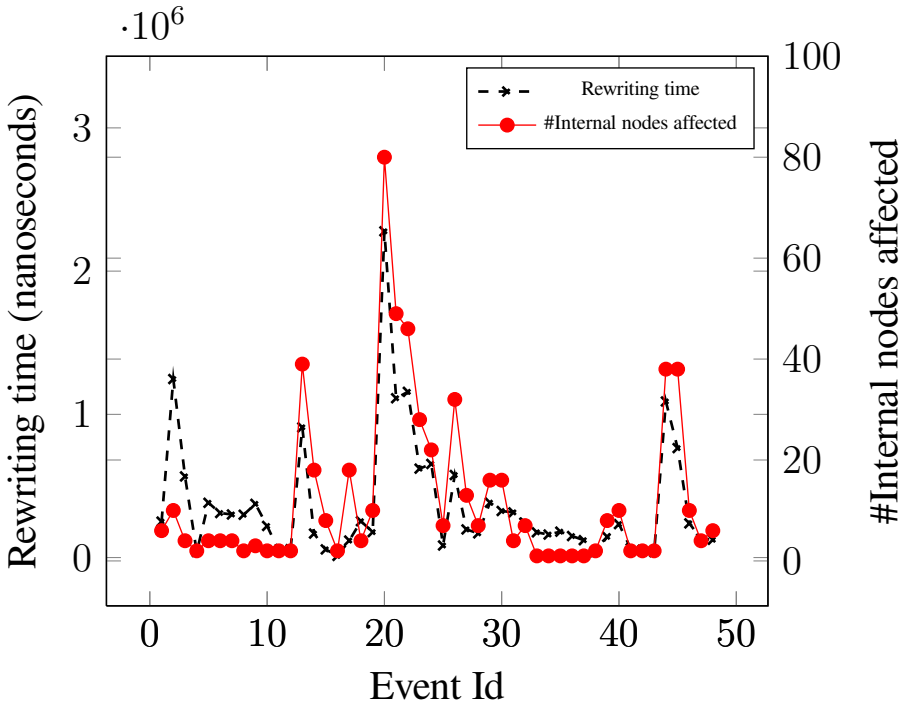

(b) Mixture

**Figure 6.2:** TPC-DS Workload 1 - Status Determination vs Nodes

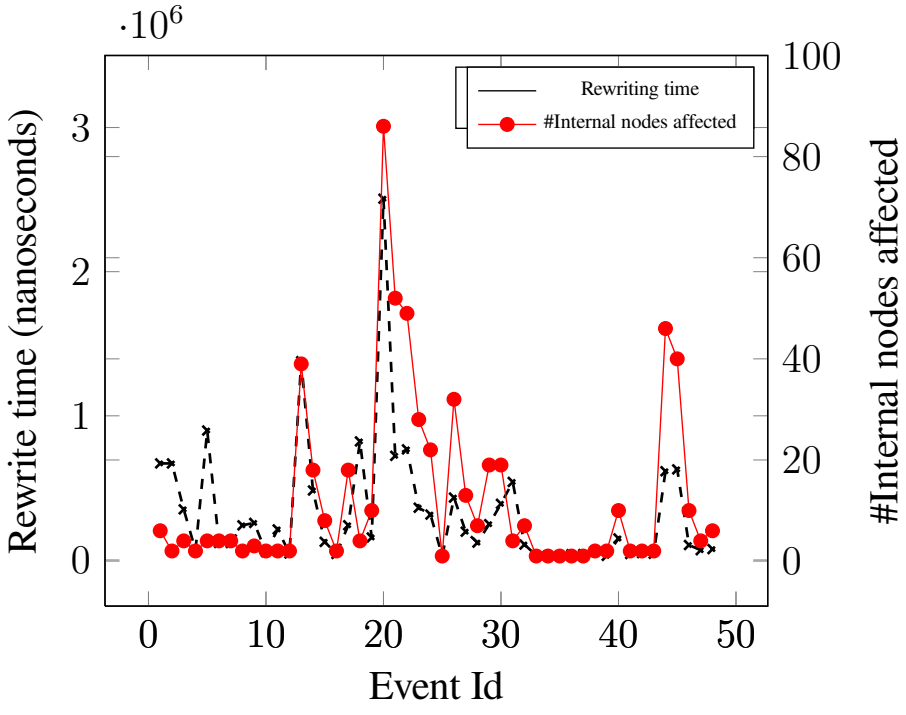
As one can see, the time spent for status determination is similar for both experiments (slightly lower for the *mixture* profile). The number of nods is practically in synchronization with the time spent with a few conspicuous exceptions, in which the time spent is disproportionately higher than the respective number of affected nodes. *We observed that this happens every time a new kind of event is executed.* Therefore, we attribute this phenomenon to the fact that the factory creating the new maestro for handling the event takes some extra time the first time it is needed; in subsequent executions the respective class information is cached and executes faster.

Similarly, there are some cases where the execution time is disproportionately lower that the number of nodes, this happens in cases of popular nodes with high fan-in (e.g., the Date_Dim relation or the Customer.C_Customer_SK attribute are used too often by many queries). In these cases, although the number of internal nodes are high, the number of modules is similar to the average (in other words, the node has a deep effect in many internal nodes of the same module); as the nodes are retrieved per module by the system, this explains why the time is lower than anticipated.

Concerning the rewriting time (Figure 6.3), it is clear that it highly depends on the combination of event with policy: if the event arrives on blockers, then, although there are affected nodes with a BLOCK status, there is no rewriting. Thus, although the rewriting time closely follows the status determination pattern for the *propagate all* profile, it is completely in disarray with it for the *mixture* profile.




(a) Propagate




(b) Mixture

**Figure 6.3:** TPC-DS Workload 1 - Rewrite vs Nodes

**Profile and rewriting times**. Although the profile has small effect on the status determination time, the rewriting times are quite different. We perform a focused study on the behavior of the rewriting times and depict the rewriting time for the two profiles in Figure 6.4. Again, the horizontal axis presents the sequence of the executed events, and the vertical axis the rewriting time for each profile (in nanoseconds). The blue dotted line concerns the *propagate all* profile and the solid red line concerns the *mixture* profile.

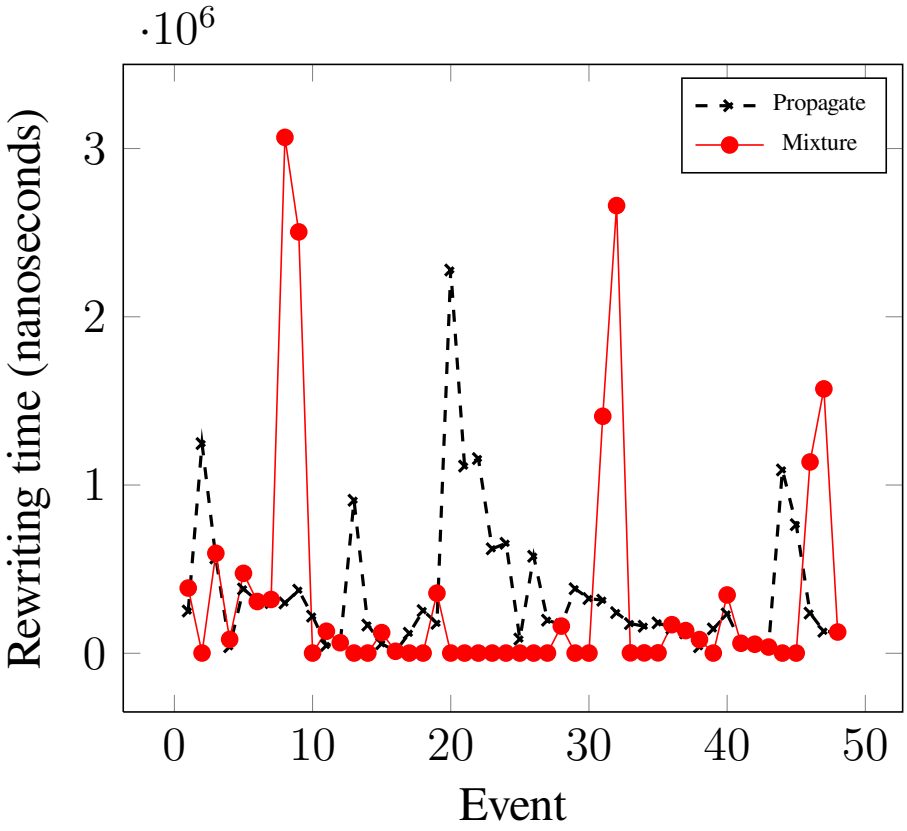



**Figure 6.4:** TPC-DS Workload 1 - Rewrite times - Propagate vs Mixture

There are two specific phenomena that present different behavior in the execution times. The first of them, concerns "solid red" peaks contrasted to "dotted blue" low-lands: i.e., we observe high times in the *mixture* profile compared to average times in the *propagate all* profile. This is attributed to view cloning: in all these cases, the combination of blocker and an adapting query, caused a view to clone. This costs highly in execution time. As the *propagate all* profile does not have blockers, there is no cloning at all. On the other hand, there are several occasions where "dotted blue" peaks are contrasted to "solid red" floors: in this case, it is all blockers that veto a change in a relation. Vetoed relation changes have no adaptation cost, as we do not go for multiple versions for (materialized) relation data (as opposed to multiple versions for view

definitions that have no storage or migration cost).

## 6.4 Discussion/Summary

A first observation on our experiments confirms the benefits introduced by our method concerning the effort performed by the application developers and administrators of the ecosystem. In the absence of our system, the typical developer would have to perform at least 25% of routine, useless checks to views and queries that are not related to the event at all; on average, the number of useless checks rises in the area of 90%.

A second observation has to do with the amount of rewriting: in all occasions, there have been several modules that had to be rewritten. When the propagate all policy was used the mean number of rewritings was 3.5 modules per workload. For instance, in the TPC-DS ecosystem (which is composed by 15 relations, 5 views and 27 queries) there were, on average, 4.5 rewrites of modules per event in the first workload and nearly 3 rewrites of modules per event in the second workload, whilst in the University ecosystem (which is composed by 5 relations, 2 views and 2 queries) there were on average 3.5 rewrites of modules. Naturally, when the policy profile changed to mixture the rewrites decreased. For TPC-DS ecosystem there was on average 1 rewrite in the first workload and 2 in the second workload. On the other hand, on the University ecosystem there were nearly 2 rewrites per event.

In terms of time, all the experiments show a completion of the tested changes as fractions of a second; specifically, the average times range in the area of 0.6 to 0.7 millisecond, whereas the maximum times do not go beyond 5 millisecond.The least time demanding part of our work is the Path Check algorithm, although it is highly impacted by the existence of blocker nodes. Path Check has as main task to mark the affected nodes of a blocker path to retain their form and to clone themselves in order their clone to accept the change, which in our experiments happened within some thousands of nanoseconds.

Although the time needed to perform impact assessment and rewriting is not significant, if we inspect the way different modules respond to different events, it is clear that the time taken to perform an event can vary a lot as a result of the popularity of a module as data-provider with its policy on the event. As Figure 6.4 depicts, when we use the mixture policy profile we can see that a number of events that is associated with relation changes takes no time at all. This happens because these events are vetoed at some point and we do not perform any kind or rewriting on blocking events of relations (e.g. event number 20, that says that relation Date_Dim will acquire a new attribute). The same event, when the *propagate all* policy profile is used, takes quite some time. This happens because the Date_Dim relation has a high in-degree and has to inform many consumers. On the other hand, there are some events that their rewrite time takes

quite a while in the mixture policy profile. This happens when a module is cloned, in order to keep the old path for the blocker module and to adapt the change for the non blocker modules it provides. For instance, event number 9 that says that view Customer_total_ret will rename itself. Here Q1 blocks the event while Q2 accepts it.

As expected, excessive peaks in impact assessment and rewriting time concern modules with high fan-in of dependent modules; these are clearly candidates where evolution should rather be blocked.

# Chapter 7

# Related work

In this chapter we are going to see other approaches that pertain to the adaption of database ecosystems in the presence of events, we will discuss their characteristics and how we relate to them. We structure our discussion along two groups of works. The first group of papers deals with efforts in the area of modeling and managing the evolution of data centric ecosystems. The second group of efforts are concerned with the rewriting of views and schema mappings in the presence of changes. Before we proceed, we have to mention a recent survey [HTR11] that done by Hartung, Terwilliger and Rahm that covers the management of schema evolution in three areas: XML, ontologies and relational data. This survey concentrates the commercial and academic attempts of relational schema evolution tools and summarizes for what each one offers to its users. Although there is a vast number of efforts in the areas of conceptual modeling, XML and ontologies, concerning the management of changes, we constrain our discussion to works that pertain to the scope of this thesis and refer the interested reader to [HTR11] for a detailed discussion of these areas.

## 7.1 Related work on data centric ecosystems

The current version of Hecataeus [PVSV10], is the one on which our work is based on. The provided framework is able to handle a great variety of evolution changes by transforming them into graph operations and by the policies that the authors introduced, it manages to regulate in which way the parts of the architecture graph are affected by evolution changes. The authors in [PVS+08] introduced an early form of policy annotation language and a status determination mechanism for the architecture graph, trying to perform an impact assessment on real-world ecosystem scenarios, while in [PVSV09] they provide an improved form of language for policy

annotation, that we based our approach on. The authors in [PVS11] provide the new model of the architecture graph and a new message propagation algorithm, on which we based our work on Chapter 3.

Similar to that approach is PRISM++, a tool of Curino, Moon, Deutsch and Zaniolo [CMDZ10] that lets the user define his policies about imminent changes. The authors use ICMOs (Integrity Constraints Modification Operators) and SMOs (Schema Modification Operators) in order to rewrite the queries/views in a way that the results of the query/view are the same as before.

## 7.2 Related work on view rewriting

**View rewriting with replacements** Nica, Lee and Rundensteiner [NLR98] attempt to make legal rewritings of views affected by changes and they primarily deal with the case of relation deletion which (under their point of view) is the most difficult change of a schema: to attain this goal one should find valid replacements for the affected (deleted) components of the existing view. In order to achieve that, the authors of [NLR98] keep a meta-knowledge base on the semantic constraints. The algorithm of CVS, has as input the following: (a) a change in a relation, (b) old MKB entities (MKB is an hyper-graph that keeps meta-information about attributes and their join equivalence attributes on other tables) and (c) new MKB entities. Assuming valid replacements exist, someone can rewrite the view via a number of joins and provide the same output as if there was no deletion.

The main steps of the CVS algorithm are: (a) find all entities that are affected for Old MKB to became New MKB, (b) mark these entities and for each one of them find a replacement from Old MKB, using join equivalences and (c) rewrite view.

**View rewriting for data migration** Gupta, Mumick, Rao and Ross [GMRR01] attempt to redefine a *materialized* view as a sequence of primitive local changes in the view definition. On more complex adaptations, those local changes can be pipelined in order to avoid intermediate creations of results of the materialized view.

This approach uses a different algorithm for each different change in a view, in order to avoid a full re-computation of the results of the view.

The following changes are supported as primitive local changes to view definitions:

1. Addition or deletion of an attribute in the SELECT clause.

2. Addition, deletion, or modification of a predicate in the WHERE clause (with and without aggregation).

3. Addition or deletion of a join operand (in the FROM clause), with associated equijoin predicates and attributes in the SELECT clause.

4. Addition or deletion of an attribute from the GROUP BY list.

5. Addition or deletion of an aggregate function to a GROUP BY view.

6. Addition, deletion or modification of a predicate in the HAVING clause. Addition of the first predicate of deletion of the last predicate corresponds to addition and deletion of the HAVING clause itself.

7. Addition of deletion of an operand to the UNION and EXCEPT operators.

8. Addition or deletion of the DISTINCT operator.

**Rewritings for schema mappings via SPJ queries** In another line of research, Velegrakis, Miller and Popa [VMP04], deal with the maintenance of a set of mappings in an environment where source and target schemata are integrated under schema mappings. The queries that the authors of [VMP04] investigate on their approach, belong to the SPJ class and the events that they investigate are the addition and deletion of constraints and the addition and deletion of nodes. The algorithm the authors suggest, needs as input: (a) the schemata of database, (b) their mappings and (c) the event that occursand produces as output the rewriting of the mappings after the event.

## 7.3 Comparison to existing approaches

Although the state of the art provides some solutions to the problems that this thesis addresses, there are potentials for improvement, which we discuss right away.

The authors of [NLR98], through the rewriting they make, they actually block the flooding of the event at the view. This way none of the view's consumers will notice any difference. Of course, this works only in case where the deleted attributes can be replaced by other join equivalence attributes. The method of [NLR98] has to be extended in order to deal with addition or renaming of attributes of a view.

The [GMRR01] approach does not handle the flow of events via policies. On the contrary, it employs the equivalent of a propagation all policy, trying to keep the data of the materialized view intact. Whenever a view changes, a user does not know what impact this has to other views/queries that are related to that, which might lead to inconsistencies of the view's consumers.

The work of [VMP04] has as main disadvantages the facts that they only relate to the SPJ class of views/queries and their approach lacks a policy mechanism.

[CMDZ10] support the ecosystem idea, to a certain extent. The approach of [CMDZ10] is like an one hop propagation of an event on our architecture graph, from a relation (they perform

rewritings only on a relation change) to its neighbors. Also, as they mention, one of their policies (IGNORE policy) may produce inconsistent results on a query execution, but they inform the user of PRISM++ about that.

The current version of Hecataeus ([PVSV10]), lacks our implementation of rewritings.

In this thesis, on the other hand we, manage to perform valid rewritings not only of a single view or query or relation, but for the entire ecosystem, through our architecture graph model. We manage to do so, regardless of the complexity of the queries and by following the user's desires, that were set via policy annotation on the architecture graph. Moreover, compared to the previous works in the context of Hecataeus, this Thesis improves the state of the art and the implementation of the tool in the following ways:

1. New model for the architecture graph
2. Rewritings of modules
3. New graphic user interface in policy assignment and event handling
4. Project management of an ecosystem, its policy files and graphs

# Chapter 8

# Conclusions and future work

## 8.1 Conclusions

*The core result of this thesis is the provision of algorithms that perform the rewriting of affected modules to adapt to the potential event.*

This thesis extended the results of the state of the art in modeling data-intensive ecosystems and assessing the impact of changes in several ways:

- First, we have extended the modeling of ecosystem *modules* (relations, views, queries) by encapsulating their structure within input and output schemata. Previous research [PVSV09] dealt with the architecture graph as a single unified graph where software modules did not have boundaries. This separation allowed us to provide theoretical guarantees for our algorithms. At the same time, it inspired the principles of modular design into the modeling of architecture graphs.

- Second, we have described the concrete implementation of the algorithms proposed in [PVS11] for impact assessment. The modular design guarantees that the propagation of a potential evolutionary event over the graph (a) terminates and (b) is performed in a way that is independent of the internal processing of events within modules.

- Third, we have implemented a concise language for policy annotation of the ecosystem modules. We followed [PVS+08] in spirit and provided a rule-based language that (a) allows the annotation of all the nodes of the graph with default policies, (b) completely covers all the space of events (i.e., all nodes have a policy to handle *any* possible event that comes to them), and, (c) allows the users to customize reaction with policies other than the default ones for individual modules. We handle events and policies that alter the

structure and semantics of the ecosystems by modifying the schema of the database and the semantics of views.

We have implemented a method that performs impact assessment and rewriting in the presence of a change. Our method (a) follows each affected module with its topological order, process the incoming messages of the module and notify its consumers, (b) checks the modules of the architecture graph for having BLOCK status and if so, notifies the paths from the module that vetoed to the module that initiated the change to also keep their current versions, and, (c) performs the rewrite of the architecture graph, letting the user see how his ecosystem would look after the change.

We have assessed our method over two ecosystems, a small university ecosystem comprising few relations and queries and TPC-DS comprising 15 relations, 5 views and 27 queries. We measured the user benefits from our method and observed that on average the user gains up to 90% of useless checks he would have to perform on views and queries that are not related to the event at all. We also observed that depending on the policy profile, the use would have to do at least 1 module rewrite per event for the *mixture* profile, and up to 4.5 module rewrites when the *propagate all* profile is used. We also assessed the performance of our method and observed that the maximum time needed for the completion of a rewrite is 5 millisecond.

## 8.2 Future Work

Naming a few of the things that could get implemented in the near future based on the work we have done are:

- Creation of maestros for events that we have not covered.
- Creation of maestros for indexes of attributes and how they affect (depending on the type of the index: bitmap, dense, sparse or reverse) the views/queries of the ecosystem.
- Work in the evolution manager in order to accept events that start from relations but get vetoed later in status determination.
- Add representation of HAVING and ORDER-BY on architecture graph.
- Move from graph representation back to SQL.

# Bibliography


[CMDZ10] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *PVLDB*, 4(2):117–128, 2010.

[Cou12] Transaction Processing Performance Council. TPC-DS: The New Decision Support Benchmark Standard. `http://www.tpc.org/tpcds/default.asp`, April 2012.

[GMRR01] Ashish Gupta, Inderpal Singh Mumick, Jun Rao, and Kenneth A. Ross. Adapting materialized views after redefinitions: techniques and a performance study. *Information Systems*, 26(5):323–362, 2001.

[HTR11] Michael Hartung, James F. Terwilliger, and Erhard Rahm. Recent Advances in Schema and Ontology Evolution. In Zohra Bellahsene, Angela Bonifati, and Erhard Rahm, editors, *Schema Matching and Mapping*, pages 149–190. Springer, 2011.

[NLR98] Anisoara Nica, Amy J. Lee, and Elke A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Advances in Database Technology - (EDBT'98), 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings*, pages 359–373, 1998.

[PAVV08] George Papastefanatos, Fotini Anagnostou, Yannis Vassiliou, and Panos Vassiliadis. Hecataeus: A What-If Analysis Tool for Database Schema Evolution. In *12th European Conference on Software Maintenance and Reengineering, (CSMR 2008), April 1-4, 2008, Athens, Greece*, pages 326–328, 2008.

[Pre00] Roger Pressman. *Software Engineering: A Practitioner's Approach: European Adaption*. McGraw-Hill, 5th edition, April 2000.

[PVS+08] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, Konstantinos Aggistalis, Fotini Pechlivani, and Yannis Vassiliou. Language Extensions for the Automation of Database Schema Evolution. In *Proceedings of the Tenth International Conference on Enterprise Information Systems (ICEIS 2008), Volume DISI, Barcelona, Spain, June 12-16, 2008*, pages 74–81, 2008.

[PVS11] George Papastefanatos, Panos Vassiliadis, and Alkis Simitsis. Propagating evolution events in data-centric software artifacts. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *ICDE Workshops*, pages 162–167. IEEE, 2011.

[PVSV07] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, and Yannis Vassiliou. What-If Analysis for Data Warehouse Evolution. In *9th International Conference on Data Warehousing and Knowledge Discovery, (DaWaK 2007), Regensburg, Germany, September 3-7, 2007, Proceedings*, pages 23–33, 2007.

[PVSV09] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, and Yannis Vassiliou. Policy-Regulated Management of ETL Evolution. *J. Data Semantics*, 13:147–177, 2009.

[PVSV10] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, and Yannis Vassiliou. HECATAEUS: Regulating schema evolution. In *Proceedings of the 26th International Conference on Data Engineering, (ICDE 2010), March 1-6, 2010, Long Beach, California, USA*, pages 1181–1184, 2010.

[VMP04] Yannis Velegrakis, Renée J. Miller, and Lucian Popa. Preserving mapping consistency under schema changes. *VLDB J.*, 13(3):274–293, 2004.

# Short Vita

Petros Manousis was born in Ioannina in 1985. He was admitted at the Computer Science Department of the University of Ioannina in 2003, and he received his BSc degree in computer science in February 2008. Since then he worked for Siemens Enterprise Communications in Athens till September 2009, when he returned to Ioannina in a new work position at Natech SA and simultaneously started his postgraduate studies. Since 2011 he is a member of Distributed Management of Data Laboratory (DMOD). His academic interests lie in the area of software engineering and data management with a particular emphasis on database evolution.