Propagation of Evolution Events in Architecture Graphs

George Papastefanatos[1], Panos Vassiliadis[2], Alkis Simitsis[3]

[1] IMIS-Athena, Athens, Hellas
gpapas@imis.athena-innovation.gr
[2] Univ. of Ioannina, Dept. of Computer Science, 45110, Ioannina, Hellas
pvassil@cs.uoi.gr
[3] HP Labs, Palo Alto, CA, USA
alkis@hp.com

Abstract. The success and wellbeing of large organizations rely on the smooth functionality and operability of their software. Such qualities are largely affected by evolution events and changes. In this paper, we are dealing with handling evolution events in data management systems. In particular, we consider a data-centric ecosystem that captures relational tables, along with their schemata and constraints, as well as, views defined on top of them and queries (being parts of software modules that are either internal to the database, e.g., stored procedures, or external software applications that access the database). We also consider policies that dictate the response of a software module to a possible event. We investigate the impact of such events to the database and present a graph-based mechanism to control propagation of events. We formally show that this mechanism terminates and that every database construct is annotated with a single status, regardless of the sequence of messages that the node receives.

Keywords: database evolution, confluence, event propagation.

1. Introduction

The success and wellbeing of large organizations rely on the smooth functionality and operability of their software. Such qualities are largely affected by evolution events and changes. The problem we are dealing with in this paper involves the identification and regulation of schema evolution impact in complex data-centric ecosystems and can be summarized as follows.

We start with the Architecture Graph of a data-centric ecosystem that captures relational tables, along with their schemata and constraints, as well as, views defined on top of them and queries (being parts of software modules that are either internal to the database, --e.g., stored procedures, or external software applications that access the database). Evolution changes affecting the database structure are mapped to graph operations on the nodes of Architecture graph. Then, the graph is annotated with policies that dictate what is the response of a software module to a possible event

(e.g., when one of the database's tables that acts as a provider of a view is to be deleted, the view can be annotated with a policy that vetoes the deletion).

Having this background we test the impact of a potential event to the graph. The main mechanism for achieving that is message propagation: every time a node receives an event, it (a) determines which policy rules apply for this event, (b) assumes the appropriate status based on these rules, and (c) notifies its neighbors for the event (if necessary) via the appropriate messages that act as events to their recipients. Hence, when a potential event is submitted, the graph must be annotated with statuses that report on whether an event is affecting a node or not, and in the case that it does, what is the actual action to be taken for the affected node. Actions imposed on affected nodes may in turn generate evolution events that are propagated as new messages towards the rest of the dependent graph structures.

Therefore, briefly, we work as follows. *Given an evolution event e over a node of the Architecture Graph v, how do we guarantee that (a) the propagation of events terminates and (b) that every node is annotated with a single status, regardless of the sequence of messages that the node receives?*

A first attempt to the problem can be found in [8]. However, that attempt focuses on a simpler data model that did not prevent multiple messages arriving at the same node and, due to this shortcoming, it cannot guarantee confluence of the evolution process. Here, we solve this issue by framing change messages within high level constructs (such as views) before they are freely flooded over the whole ecosystem's graph. The benefits of this process are as follows. We achieve localization of decisions and guarantee satisfactory handling of event transactions. Working like this, we are also able to achieve nice properties, like confluence.

Outline. The rest of the paper is structured as follows. Section 2 discusses modeling issues. Section 3 and 4 present the message propagation mechanism and some theoretical results, respectively. Section 5 discusses the related work and Section 6 concludes the paper.

2. Background Modeling

In this section, we built upon our model of the architecture graph [7] and extend it in order to guarantee a safe, confluent mechanism for message propagation. Here, we briefly present its main modeling components and highlight how this model is extended. *In a nutshell, the main difference with* [7] *lies in the structure of views and queries: here, views and queries are containers of nodes, encapsulated between the input schemata and the output schema of a view/query*. Previously, we did not consider the input and output schemata as first class citizens of our model.

2.1 Architecture Graph

Our modeling technique represents all the aforementioned database constructs as a directed graph **G=(V,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 (a) relations, (b) views, (c) 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 subgraph of the Architecture graph is assumed. For example, the attributes and local (e.g., PK, NotNull,etc.) 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. For reasons of clarity, we avoid referring to these relationships explicitly, unless this is absolutely necessary.

Relations, **R**. Each relation $R(\Omega_1,\Omega_2,\ldots,\Omega_n)$ in the database schema, either a table or a file (it can be considered as an external table), is represented as a directed graph, which comprises: (a) a *schema node*, R, representing the relation's schema; (b) n *attribute nodes*, $\Omega_i \in \mathbf{\Omega}$, $i=1..n$, one for each of the attributes; and (c) n *schema relationships*, $\mathbf{E_S}$, directing from the relation node towards the attribute nodes, indicating that the attribute 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., <, >, =, ≤, ≥, !=, IN, EXISTS, ANY): (a) Ω op constant; (b) Ω op Ω′; and (c) Ω op Q. (Ω, Ω′ 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 subgraph corresponding to the semantics of the query, and,
(e) *attribute nodes* belonging to the various input schemata and output schema 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`, each of which is eventually mapped to a subgraph.

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 relationships*, $\mathbf{E_M}$, directing from the output attributes towards the input schema attributes.

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 relationships*, $\mathbf{E_F}$, directing from the nodes of the appropriate input schemata towards the relation nodes. 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 $\mathtt{GB} \in \mathbf{GB}$, 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.

The representation of the ORDER BY clause of the query is performed similarly.

Functions, F. Functions used in queries are integrated in our model through a special purpose node $F_i \in \mathbf{F}$, 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 as nodes and edges denoting 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 $\mathbf{V_S} \subseteq \mathbf{R} \cup V \cup Q$ comprising the graph's module nodes and $\mathbf{E_S} \subseteq \mathbf{E_F}$ comprising pairs of providers and consumers as *from-relationship* edges, $\mathbf{E_F}$.

Example. The following example (Fig. 1) 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.

On top of this database, 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

We have omitted all constraints (e.g., primary and foreign key) as well as *map-select* edges from the figure to avoid overcrowding it. The *map-select* edges can be deduced from the names of the attributes

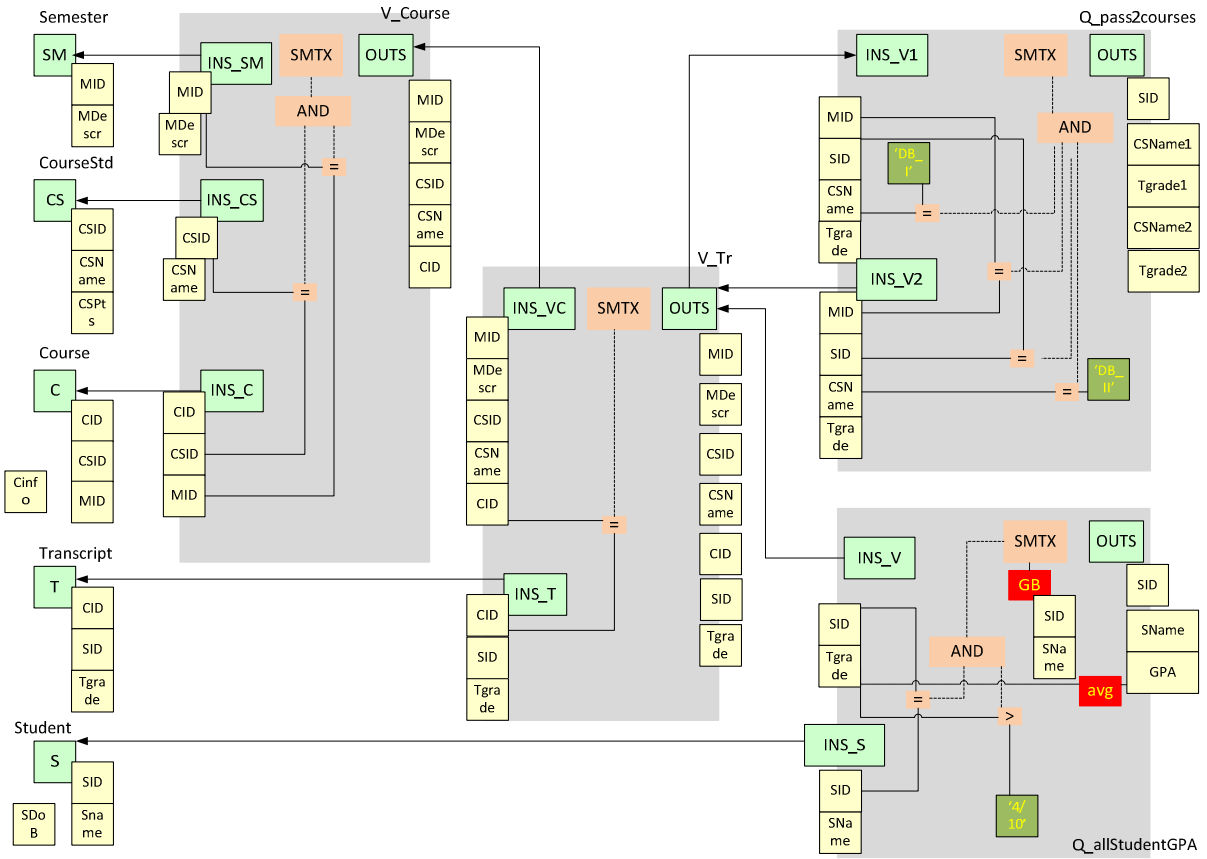

Fig. 1. Architecture graph of the Reference example

2.2 Graph Annotation with Policies

The presented graph model enables us to capture the various dependencies between the modules of the ecosystem at a most granular level. Apart from the simple task of capturing the semantics of a database ecosystem, the proposed graph model allows us to evaluate the impact of a change over the system. In [8] we have provided a subtle technique for mapping schema changes occurring at the database ecosystem to operations on the node of the graph (e.g., the addition of an attribute in a relation is mapped to an addition of a child node in the relation module). In addition, we have enriched the graph with rules, called *policies* that dictate the actions that are performed, when specific events occur on the nodes of the graph. Policies can be applied at various granularity levels on the graph, i.e., from the module level down to the level of attributes and operand nodes, ensuring that the reaction to events for all nodes in the graph [9]. Two kinds of rules are defined with respect to the semantics incurred by an event, (a) *propagate* the change, meaning that the graph must be reshaped to adjust to the new semantics incurred by the event; and (b) *block* the change, meaning that we want to retain the old semantics of the graph and the hypothetical event must be blocked or, at least, constrained, through rewriting that preserves the old semantics. For instance, the policy "On add_`attribute` to `Transcript` Then `propagate`" defined on `V_Tr.INS_T` node dictates the propagation of the addition of a new attribute in the `Transcript` relation towards the schema of the view. Simple default values and policy resolution rules can safely guarantee that all nodes can determine the appropriate policy for any event they receive. We refer the interested reader to [8], [9] for a thorough discussion.

2.3 Message propagation and Status Resolution

Whenever a hypothetical event over a node (e.g., the deletion of an attribute) is submitted to the graph, the system must ensure that (a) the event is propagated to all the nodes that are affected directly or transitively, (b) each of the affected nodes acquires the correct status, according to its annotation with policies for this event.

Policy Determination. Clearly, it would be very hard for the user to have to define a policy per event for every module of the Architecture Graph. In [9], we have defined a language where the user can dictate "default" policies both at the graph level and for the children of individual nodes, in order to avoid this effort. In fact, the language allows the user to define policies at different levels of abstraction which can be overriding one another (so, for example, if the default policy for the deletion of input schema attributes is *block*, the user can override it for the input schema attributes of a particular view). Then, a late-binding mechanism determines the winner policy for each specific node. For a most detailed description of the policy annotation and determination, we refer the interested user to [9].

Status Determination. Status determination stems from the simple application of rules. Given a finite vocabulary of events, V_E, a finite vocabulary of policies V_P and a

finite vocabulary of statuses V_S, the only thing that we need is a set of rules as function *DS:* V_E x V_P -> V_S.

Table 1. Vocabularies for events, policies and statuses of nodes

V_E	{SELF, CHILD, PROVIDER} x {ADD, DEL, UPD} x {STRUCTURE, SEMANTICS, S+S}
V_P	{BLOCK, PROPAGATE}
V_S	{SELF, CHILD, PROVIDER} x {ADD, DEL, UPD} x {STRUCTURE, SEMANTICS, S+S }

Table 1 explains that the events for which a node is notified state that either itself, or one of its children (e.g., a relation's attribute, or an attribute's constraint) or a provider is affected (by addition, deletion or update) with respect to its structure, semantics or both. The policies are either *Block* (which dictates that the event is practically vetoed at the node's scope) or *Propagate*. Lastly, the statuses assigned can have a scope indicating whether the impact of the event refers to the node itself, its ancestors within a module or its provider nodes.

Event propagation. The third part of the mechanism is the broadcasting of messages to the neighbors of a node that acquires a status for an event. Each message corresponds to a unique event occurring on the sender node and describes the event type and the status assigned to it according to the prevailing policy. Each message is processed locally, inside a module and may trigger one or more events for further propagation to the consumer modules.

For example, the deletion of an attribute that participates in the SELECT and WHERE clauses of a view, generates a new message for the consumers of the view; this message encodes the modification of the view's structure and semantics.

3. Message propagation mechanism

At the high level, the graph nodes form a directed acyclic graph of 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 module with its topological order and we check whether there are incoming messages for it. 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).

We examine, now, the protocol for handling of the events within each module as well as the structure and contents of the outgoing message to the consumer nodes of an affected node; this is the topic of this section.

There are four kinds of nodes involved in the propagation mechanism.

- *Input schemata nodes*, which are the kind of nodes that receive notifications for changes from other modules.
- *Internal node*s, which are: (a) possibly affected by the changes to the nodes of the input schema and (b) amenable to evolution events by the users (e.g., a user altering the selection condition or the grouper attributes of a view).
- *Output schema nodes*, which are the only nodes who emit messages to their consumer modules for the possible modification of their component.
- *Semantics nodes*, which determine whether the semantics of a component are the same or not and inform the output schema nodes for further propagation.

The requirements that we want to address regarding the message handling within a module are that each event must affect the appropriate nodes and that every node must be visited and processed (i.e., its status must be determined) once per message. Hence, we introduce a process mechanism with the following characteristics:

(a) Messages arriving at a node are propagated to all of its consumers (i.e., adjacent nodes connected with an incoming edge to this node) *according to the type of event* that they encode (e.g., the addition of an attribute is propagated only to semantics and output nodes, whereas the deletion of an attribute is propagated to attribute nodes). We describe this mechanism in the following sections.

(b) For each event initiated by the input schema or the user, we identify the affected subgraph of the module according to the protocol mechanism as described in the following sections. For identifying the subgraph, we process each event by executing the protocol mechanism and assuming that no policies constrain the flooding process. The produced subgraph contains only these nodes potentially affected by this event.

(c) Given that each identified subgraph is acyclic (see theorem 3), we again perform a second execution of the protocol, starting from the input schema (or the node affected explicitly by the user) and visiting each node in a topological order of the subgraph. According to the policy defined, an event processed on a node generates one or more messages, which are enqueued in the message list of all of its consumers. The next node to be processed is the next in the topological order of the identified subgraph. The mechanism guarantees that each node is processed once, after all possible messages have arrived at it.

Next, we present the message handling mechanism for each class of nodes. The general structure of the event propagation mechanism is presented in Figure 2.

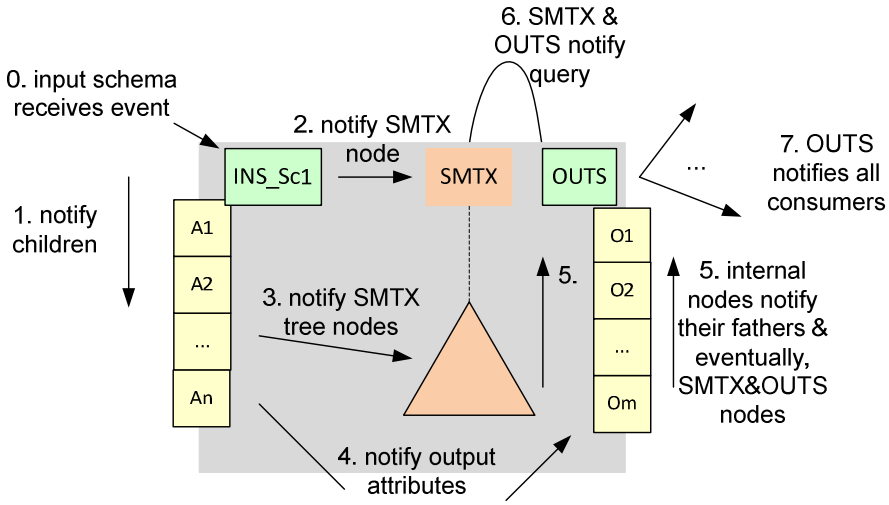

Fig. 2. Event propagation mechanism within modules

Input schema nodes. An input schema receives messages from the output schema nodes of the providing relation / view containing occurred events on the provider's structure or semantics. For instance, the output schema of a relation module can report the following events to the input schema node of a query accessing this relation:

(a) The relation is renamed or deleted.
(b) Attributes are added/ deleted / updated (renamed or modified).
(c) Constraints are added/ deleted / updated (renamed or modified). Events occurred on relation constraints eventually generate messages stating the semantic change of the relation.

For any of the above events a message is constructed and received by all input schema nodes accessing the relation. The detail mechanism that is triggered by the input schema node when receiving such a message is as follows:

(i) The correct policy (based on the type of the received message) is determined for the receiving input schema node.
(ii) The rule dictating the policy is fired and the appropriate status is assumed. In the case of propagation, the node assumes a status for adjusting to the event, whereas in the case of block policy, the node takes a status for blocking the event. For example, in the case of an incoming message for the addition of a new attribute, for which the input schema retains a propagate policy, the input schema node is assigned with a status for adding a child.
(iii) For events referring to the input attributes (e.g., deletion of an attribute at the provider's schema, renaming, domain modification, etc.) the appropriate input attribute nodes of the input schema are notified.
(iv) The input schema node propagates a message containing changes on the semantics of the provider module directly to the semantic node of the current module – if such changes exist; otherwise no such action is taken.

(v) The input schema node propagates a message for addition of children towards the output schema node of the module and (if any) to the group by node via the semantic node.

Observe that each input schema has exactly one provider (i.e., the output schema of the provider module. Hence, it can receive exactly one message that triggers the evolution handling mechanism in every module. In other words, a module can receive, at most, as many event handling messages for the same original event as its input schemata. An alternative way to start the mechanism is by a user applying a change at the module, which again triggers exactly one "input" message possibly at an internal node.

Internal nodes. These can be either attributes in the input/ output schemata of a module, or logical components of the semantics node of a module, like a function node, an operand or a constant node, group by node, etc.

Intra-module nodes can receive messages either from (a) their father (e.g., an input schema node notifies that a specific input attribute must be deleted), (b) from their provider nodes (e.g., an output attribute node or an operand node is notified by its provider attribute in the input schema for its deletion), (c) one of its children or lastly (d) explicitly by the user who triggers the modification of the node itself.

The message propagation for the nodes of this category mainly notifies all their consumers on what is happening to them as well as notifying the semantics node on whether the semantics of the component change. Therefore, the mechanism for each such node is as follows:

(i) – (ii) The first two steps are like those of the input schema nodes.

(iii) If the node has children and receives a notification from its father or if it initiates the event, then its children are notified too. This mainly applies to operand nodes in composite conditions at views and queries or relations' attributes having constraints (e.g., conditions) as children.

(iv) If the node notified by one of its providers or one of its children, the father of the node is notified, too. This covers the case where a user triggers an event in the contents of a view (e.g., deletion of a condition) or a relation (e.g., modification of an attribute), so that the event would be also propagated upwards to the module node.

(v) In all cases, the node consumers (if any) are notified too. This covers the case where an input attribute is changed, so that the event is propagated towards all nodes (i.e., output attributes, conditions, functions, group by attributes) that refer to this attribute.

Observe that this way, every node notifies its consumers, and every node does not receive a message for the same event more than once per edge. The key here is that there is a flow of the messages, either from an ancestor node towards its descendants, or from an intermediate node towards both its ancestors and its descendants. Since cycles do not exist, every node receives each message exactly once, except for binary nodes (e.g., '=' or 'AND' nodes in constraint trees that might receive a status from both their edges). However, due to the topological sorting of the tree, if an intermediate node in the semantics tree receives a first notification from one of its two edges, it is possible to check whether another notification is also pending before deciding its final status. At the end of this process, the semantics node receives one

message (either a contingency blocking veto, or a propagation message) for a potential change in the subtrees of the Where or the Group By nodes.

Semantics nodes. A semantics node receives messages either from the input schema of the module for messages containing changes on the semantics of the provider modules or from its children. The mechanism that is triggered by the semantics node when receiving such messages is as follows:

(i) – (ii) The first two steps are like those of the input schema nodes.

(iii) The semantic node propagates a message for addition of children towards (if any) the group by node.

(iv) The semantics node propagates all other messages coming from either the input schema node (e.g., for changes in the semantics of a provider module) or its children (i.e., for changes in the semantics of the module itself) to the output schema node of the module.

Output schema nodes. The output schema is responsible for establishing the overall status of the module. An output schema node can receive messages from the semantics node regarding semantic changes in the module, from the input schema for additions of attributes or from one of its children for changes referring to the exposed structure of the module. The following mechanism is applied for handling a received event signal:

(i) – (ii) The first two steps are like those of the input schema nodes.

(iii) The father of the output schema node, i.e., the module's node, is notified too. Whenever the module's node gets a notification from the output schema it acquires the right status (i.e., *block* if a veto has been fired or the appropriate status in any other case).

(iv) Except for the case the assigned status is *block*, all consumers (input schemata) of the output schema node are notified with a message announcing the module's status

Table 2: Overview of message propagation for each kind of node in a module

Messages arrive from	Type of node	Messages propagated to
{provider's output schema}	Input schema	{children, semantics, output schema}
{father, provider, children, user(self)}	Internal Nodes	{children (if any), consumers, father}
{input schema, children}	Semantics	{output schema}
{semantics, children, input schema}	Output schema	{consumers' input schema, module}

Example: We, illustrate the propagation mechanism by examining two specific evolution events on the graph of Fig.1: (a) the addition of a new attribute to the `Transcript` relation, namely `ExamYear`, that represents the year that the student has taken the exam on each course and (b) the removal of attribute `MDescr` from the select clause (i.e., output schema) of `V_Course` view. We assume that both changes are explicitly invoked by the user and for each of them, we hold the nodes that are

visited by the algorithm, the kind of messages that arrive to these nodes, the status that is assigned on them, the messages that they emit and lastly the forward nodes that they inform. We, also, assume that *propagate* policy is assigned on all visited nodes and thus termination of the mechanism is not caused by a block policy.

(a) *Addition of `ExamYear (EY)` attribute to `Transcript` table*: The message propagation for this event is presented in Table 3. The message for the addition of `EY` on the `Transcript` node, results in assigning the appropriate status for adding `EY` as a new child. Since the policy is *propagate*, an identical message is created and the input schema node `V_TR.INS_T` connected with the `Transcript` node is visited. The `V_TR.INS_T` node adapts the event and informs the output schema node for the addition. The affected subgraph for this event according to our mechanism comprises nodes {`V_TR.INS_T, V_TR.OUT_S,V_TR`} which are visited in this order.

The `V_TR.OUT_S` node propagates, in turn, the event towards all input schema nodes referring to the `V_Tr` view. For the `Q_pass 2courses` (`Q1`) query, each input schema node (i.e., `Q1.INS_V2` and `Q1.INS_V1`) receives a distinct message for the attribute addition. These messages are propagated towards the query output schema Q_1.OUT_S as two separate events. The message propagation terminates on the output schema nodes of the two queries (see Figure 1 too), `Q1.OUT_S` and `Q2.OUT_S`, as no other consumer modules exists. The output schema of the `Q_allStudentsGPA` (`Q2`) query, receives two messages for two separate events; one for the addition of the attribute in the input schema of the query and the other for the modification of the semantics as result of the incorporation of the new attribute to the group by clause of the query.

Table 3: Message Propagation for the addition of `ExamYear` to table `Transcript`

visited module	Visited Node	message arriving	Status	message emitted	next node in queue
Transcript	Transcript	AC {EY}	To_AC	AC {EY}	V_TR.INS_T
V_TR	INS_T	AC {EY}	To_AC	AC {EY}	OUT_S
	OUT_S	AC {EY}	To_AC	AC {EY}	{V_TR Q1.INS_V1,Q1.INS_V2,Q2.INS_V}
	V_TR	AC {EY}	To_AC	none	None
Q1	INS_V1	AC {EY}	To_AC	AC {EY}	OUT_S
	INS_V2	AC {EY}	To_AC	AC {EY}	OUT_S
	OUT_S	AC {EY}	To_AC	AC {EY}	Q1
	Q1	AC {EY}	To_AC	AC {EY}	None
Q2	INS_V	AC {EY}	To_AC	AC {EY}	{SMTX, OUT_S}
	SMTX	AC {EY}	To_MS	AC{EY},MS	{GB, OUT_S}
	GB	AC{EY}, MS	To_AC	AC{EY}	None
	OUT_S	AC{EY}, MS	To_AC, To_ MS	AC{EY}, MS	Q2
	Q2	AC{EY}, MS	To_AC, To_ MS	none	None

Legend: AC: Add_Child,Q1: Q_pass2courses, Q2: Q_allStudentsGPA, MS: Modify_Semantics

Table 4: Message Propagation for the deletion of **`MDescr`** from **`V_Course`** view

visited module	visited node	message arriving	status	message emitted	next node in queue
V_Course	MD	DS	To_DS	DS	OUTS
	OUTS	DS	To_DC	DC{MD}	V_TR.INS_VC
V_TR	INS_VC	DC{MD}	To_DC	DC{MD}	INS_VC. MD
	INS_VC.MD	DC{MD}	To_DS	DS	OUT_S. MD
	OUT_S.MD	DS	To_DS	DS	OUT_S
	OUT_S	DS	To_DC	DC{MD}	V_TR
	V_TR	DS	To_DC	DC{MD}	none

Legend: DS: Delete_Self, DC: Delete_Child, MDescr: MD

(b) *Deletion of attribute `MDescr` from `V_Course` view.* The removal of attribute `MDescr(MD)` from the select clause of `V_Course` view starts on `MDescr` node of output schema node `V_Course.OUT_S` (Table 4). The message is first propagated towards its father, namely `V_Course.OUT_S`, which is assigned with a status for deletion of one of its children, and then input schema node `V_TR.INS_VC` is informed about the deletion. `V_TR.INS_VC` is assigned with the same status and propagates the message to the specific child node to be deleted. Attribute `MDescr` of `V_TR.INS_VC` informs, in turn, the consumer attribute belonging in the output schema of the view. The propagation terminates on the output schema node `V_TR.OUT_S` since no other consumers exist for the specific attribute.

4. Theoretical Guarantees

In this section, we present the theoretical guarantees for the correct execution, termination and confluence of the aforementioned protocol mechanism on the architecture graph. We examine and prove these properties both at the summary graph, i.e., at the intermodule level (theorems 1-3), as well as within each module (theorem 4).

4.1 Guarantees at the intermodule level

In this subsection, we prove that the mechanism for message propagation works correctly at the summary or, *intermodule* level. We assume that each module responds correctly to a given event; we prove this property in the subsequent subsection.

Theorem 1 (termination). The message propagation at the intermodule 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 at most one message for each session to every one of its neighbors; (b) the graph is finite. Since both assumptions hold, the algorithm terminates. □

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

Proof: At the summary level, each input schema of a consumer module receives the status and the output schema structure of its provider module. The **topological ordering** of the graph guarantees that whenever a module is considered, all its providers have already been processed. So, all that remains is to prove that once all notifications from the module's providers are in place, the module will uniquely acquire a status. This is proved in Theorem 4. □

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. From 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; therefore, they can safely ignore any notification. □

4.2 Guarantees at the intramodule level

In this subsection, we prove that once an event arrives at a module, the module responds to the event and annotates the output schema with the correct status.

Theorem 4 (termination and correctness). The message propagation at the intramodule level terminates and each node assumes a status.

Proof: At the intra-module level, for the termination of the mechanism, we must prove that each constructed subgraph per event type is a directed acyclic graph. For the correctness of the mechanism we require that every node is processed once (and thus assigned with a status) for all messages arriving at a module per session. The latter can be satisfied when the determined subgraph can be topologically sorted and traversed. Thus, for both requirements we must prove that the subgraph that is constructed per event type has no cycles. We cover the following types of messages arriving at the module:

- Change in semantics of provider: the message arrives to the input schema node and is propagated to the semantics node. The affected subgraph comprises the following nodes and directed edges in topological order:

{input schema→semantics→output schema→module}[1]. No cycles detected.

- Internal change in the semantics of a module (e.g., a user deletes a part of the condition expression of a view): the semantics node is eventually notified from the upwards flow of messages in the semantics tree and the children are notified from the downwards flow of messages.
  - For the case that a condition node is modified, subgraph comprises {internal node→up condition tree→semantics→output schema→module}, {internal node→down condition tree}. No cycles detected.
  - For the case that a grouping attribute is modified, subgraph comprises: {GB Attributes →GB→semantics→output schema→module}
- Deletion in the structure of the input schema: all affected nodes in the tree of the condition part are notified via the operand relationship edges; all group by and output schema are notified via the map-select edges. Subgraph potentially (if group by part exists) comprises:
  {input schema→input attributes},
  {input attributes→condition tree→semantics},
  {input attributes→GB attributes→GB node→semantics},
  {input attributes→output attributes→output schema}
  {semantics→output schema} and
  {output schema→module}. No cycles detected.
- Addition in the structure of the input schema: a message is sent to the output schema and to the semantic node for informing the group by node (if any). Subgraph potentially comprises:
  {input schema→semantics},
  {input schema→output schema},
  {semantics→GB node},
  {semantics→output schema},
  {output schema→module}. No cycles detected
- Deletion of in the input schema overall (the provider dies overall too): the deletion is correctly propagated from the messages sent by all the child nodes of the schema.
- Change in structure (deletion or addition) and semantics of a provider. When messages arriving at an input schema node contain changes both at the structure and the semantics of the provider module, the subgraph is the union of the subgraphs corresponding to each case. Thus, for attribute addition and change in provider semantics, the subgraph is:
  {input schema→semantics},
  {input schema→output schema},
  {semantics→GB},
  {semantics→output schema→module}. No cycles detected
  For attribute deletion and change in provider semantics, the subgraph is:
  {input schema→semantics},
  {input schema→input attributes},
  {input attributes→condition tree→semantics},
  {input attributes→GB attributes→GB node→semantics},

[1] For ease of graph serialization we denote an edge directing from input schema towards semantics as "input schema→semantics".

{input attributes→output attributes→output schema}
{semantics→output schema} and
{output schema→module}. No cycles detected

In all cases, at the end of the process, the output schema (and eventually the module itself) has knowledge (a) of what happens to their children and (b) what happens to module and can pass this information to the next consumer. □

Theorem 4 dictates that all nodes at the intramodule level will be processed once and assigned with a status. The processing order is predetermined by the type of event arrived at the module and the propagation mechanism terminates at the module node, which denotes the overall status of the module.

The status assigned to each node is determined by the type of node, the type of event encoded in the message arriving at the node and the type of policy defined on the node; thus, if only one message arrives at a node, a unique status can be easily determined. However, in the case that a node receives two or more messages (i.e., from many providers) containing different events, a unique status must first be resolved for these different events and then propagated to next nodes. In the next proposition we show that all internal nodes eventually obtain a unique status per session, regardless of the number of different messages arriving at them.

Proposition 1 (unique status). All internal nodes visited by the algorithm will obtain a unique status according to the defined policy and the types of events encoded in the received messages.

Proof: We distinguish the following types of nodes: input schema node, input attributes, semantics node, GB node, grouping attributes, nodes in condition tree (condition nodes and constants), output attributes, output schema node and lastly, module node. According to theorem 4, for all cases of events, the types of nodes which can potentially receive two or more different messages in a single session are the output schema and the semantics nodes. All other nodes have either one provider in each subgraph derived by the mechanism (e.g., in the case of attribute addition the GB node receives a message from the semantics node, whereas for attribute deletion it receives a message from one of the grouping attributes), or multiple arrived messages are of the same type (e.g., a condition node receives two messages for the deletion of both operand nodes). Unless the policy defined on the node is *block* (the status is always resolved as block), the status is uniquely determined according to the type of event. A semantics node however can receive at the same time a message from the input schema (e.g., for a change in the providers' semantics or the addition of an attribute) and one or more messages from its children (e.g., for the deletion of a condition or a grouping attribute). In all cases the resolved status for *propagate* policy is the same, namely "to modify semantics", and thus no ambiguous statuses can be assigned to the semantics node. Output schema, on the other hand, can potentially receive a message from the input schema for attribute additions, a message from the semantics node for the update of the module semantics, and a message from its children for the deletion of an output attribute. For propagate policy, the resolved status depends on the received message and can denote modification in semantics, structure or both (e.g., "to modify semantics and add attribute X", or "to modify semantics and delete attribute Y", etc.). In all cases, statuses are uniquely identified in all nodes within a module, regardless of the number of messages received.

5. Related Work

Schema evolution is a long-term problem in database research, addressed each time under the specific characteristics and operations of the approached data model. In [11] one of the earliest surveys on schema versioning and evolution is presented, whereas a categorization of the overall issues regarding evolution and change in data management is presented in [10]. Evolution related approaches have been also proposed for the OO paradigm [13] and DW configurations [3, 4], as well. Relational schema evolution and versioning are revisited in [5], where the authors introduce a technique for publishing the history of a relational database in XML, employ a set of schema modification operators (SMOs) to represent the mappings between successive schema versions and an XML query language to efficiently address queries expressed over different versions using the mappings established by the SMOs.

Related to our approach, the problem of view adaptation after redefinition is mainly investigated in [1, 2] where changes in views definition are invoked by the user and rewriting is used to keep the view consistent with the data sources. Also, [5] deals with the view synchronization problem, which considers that views become invalid after schema changes in their definition. The authors extend SQL, enabling the user to define evolution parameters characterizing the tolerance of a view towards changes and how these changes will be dealt with during the evolution process. In this context, our work can be compared with that of [5] in the sense that policies act as regulators for the propagation of schema evolution on the graph similarly to the evolution parameters introduced in [5]. In [12] a similar framework for the management of evolution is proposed. Still, the model of [12] is more restrictive, in the sense that it is intended towards retaining the original semantics of the queries by preserving mappings consistent when changes occur. Our work is a larger framework that allows the restructuring of the database graph (i.e., model) either towards keeping the original semantics or towards its readjustment to the new semantics. In addition, we employ a detail representation and a message propagation mechanism for detecting and regulating evolution impact in complex database ecosystems.

Our rule-based propagation of schema evolution changes in architecture graphs shares some common characteristics with problems related to the definition and management of active rules in database systems. We mention the work presented in [14], where the authors formulate the problems of termination, confluence and observation in active database rules. Given a set of active rules, the authors provide formal methods for evaluating whether the execution of this set terminates, produces a unique final database state, or finally produce a unique stream of observable actions regardless of the rules' execution order.

6. Conclusions

In this paper, we focused on the problem of change propagation in database ecosystems. Based on a graph representation of database constructs and considering that this graph can be annotated with policies dictating the response of a software

module to a possible event, we investigated the impact of such events to the database and presented a graph-based mechanism to control propagation of events.

7. References

1. Z. Bellahsene, "Schema evolution in data warehouses". *Knowledge and Information Systems*, vol. 4, no. 3, pp. 283-304, May 2002.
2. Gupta, I. S. Mumick, J. Rao, K. A. Ross, "Adapting materialized views after redefinitions: Techniques and a performance study". *Information Systems J*, vol. 26, no. 5, pp. 323-362, Jul. 2001.
3. M. Golfarelli, J. Lechtenbörger, S. Rizzi, G. Vossen, "Schema Versioning in Data Warehouses", *Proc. Conceptual Modeling for Advanced Application Domains, ER 2004 Workshops (ECDM'04)* , pp. 415–428, 2004.
4. C. Kaas, T. B. Pedersen, B. Rasmussen, "Schema Evolution for Stars and Snowflakes". *Sixth Int'l Conference on Enterprise Information Systems (ICEIS'04)*, pp. 425-433, 2004.
5. Moon, H.J., Curino, C.,Deutsch, A., Hou, C.Y., Zaniolo, C. Managing and querying transaction-time databases under schema evolution. In VLDB' 08, pp. 882-895, 2008.
6. Nica, A. J. Lee, E. A. Rundensteiner, "The CSV algorithm for view synchronization in evolvable large-scale information systems". *Proc. Sixth International Conference on Extending Database Technology (EDBT'98)*, pp. 359-373, 1998.
7. G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou, "What-If Analysis for Data Warehouse Evolution". *Proc Ninth International Conference on Data Warehousing and Knowledge Discovery (DAWAK'07)*, pp. 23-33, 2007.
8. G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou. *Policy-Regulated Management of ETL Evolution*. In Springer Journal on Data Semantics, vol. XIII, pp. 146–176, 2009.
9. G. Papastefanatos, P.Vassiliadis, A.Simitsis, K.Aggistalis, F.Pechlivani, Y.Vassiliou, "Language Extensions for the Automation of Database Schema Evolution". *In 10th International Conference on Enterprise Information Systems (ICEIS '08)*,2008.
10. J.F. Roddick et al, "Evolution and Change in Data Management - Issues and Directions". *SIGMOD Record*, vol. 29, no. 1, pp. 21-25, 2000.
11. J.F. Roddick., "A survey of schema versioning Issues for database systems". *Information Software Technology J.*, vol. 37, no. 7, 1995.
12. Y. Velegrakis, R.J. Miller, L. Popa, "Preserving mapping consistency under schema changes". *VLDB J.*, vol. 13, no. 3, pp. 274-293, 2004.
13. R. Zicari, "A framework for schema update in an object-oriented database system". *Proc. Seventh International Conference on Data Engineering (ICDE'91)*, pp. 2-13, 1991.
14. A. Aiken, J. M. Hellerstein, J. Widom: Static Analysis Techniques for Predicting the Behavior of Active Database Rules. ACM Trans. Database Syst. 20(1): 3-41 (1995).