Automating the Adaptation of Evolving Data-Intensive Ecosystems

Petros Manousis[1], Panos Vassiliadis[1], and George Papastefanatos[2]

[1] Dept. of Computer Science University of Ioannina (Hellas)
{pmanousi,pvassil}@cs.uoi.gr
[2] Research Center "Athena" (Hellas)
gpapas@imis.athenainnovation.gr

Abstract. Data-intensive ecosystems are conglomerations of data repositories surrounded by applications that depend on them for their operation. To support the graceful evolution of the ecosystem's components we annotate them with policies for their response to evolutionary events. In this paper, we provide a method for the adaptation of ecosystems based on three algorithms that (i) assess the impact of a change, (ii) compute the need of different variants of an ecosystem's components, depending on policy conflicts, and (iii) rewrite the modules to adapt to the change.

Keywords: Evolution, data-intensive ecosystems, adaptation.

1 Introduction

Data-intensive ecosystems are conglomerations of databases surrounded by applications that depend on them for their operation. Ecosystems differ from the typical information systems in the sense that the management of the database profoundly takes its surrounding applications into account. In this paper, we deal with the problem of facilitating the evolution of an ecosystem without impacting the smooth operation or the semantic consistency of its components.

Observe the ecosystem of Figure 1. On the left, we depict a small part of a university database with three relations and two views, one for the information around courses and another for the information concerning student transcripts. On the right, we isolate two queries that the developer has embedded in his applications, one concerning the statistics around the database course and the other reporting on the average grade of each student. If we were to delete attribute C_NAME, the ecosystem would be affected in two ways : (a) *syntactically*, as both the view V_TR and the query on the database course would crash, and, (b) *semantically*, as the latter query would no longer be able to work with the same selection condition on the course name. Similarly, if an attribute is added to a relation, we would like to inform dependent modules (views or queries) for the availability of this new information.

The means to facilitate the graceful evolution of the database without damaging the smooth operation of the ecosystem's applications is to allow all the involved stakeholders to *register veto's or preferences*: for example, we would

W. Ng, V.C. Storey, and J. Trujillo (Eds.): ER 2013, LNCS 8217, pp. 182–196, 2013.
© Springer-Verlag Berlin Heidelberg 2013

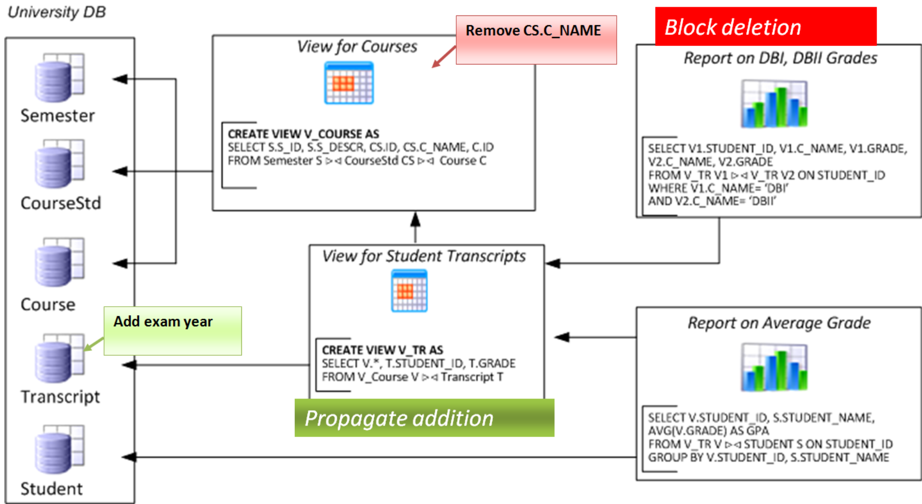

Fig. 1. Managing the adaptation of a University-DB Ecosystem

like to allow a developer to state that she is really adamant on retaining the structure and semantics of a certain view. In our method, we can annotate a *module* (i.e., relation, view or query) with a *policy* for each possible event that it can withstand, in one of two possible modes: (a) *block*, to veto the event and demand that the module retains its previous structure and semantics, or, (b) *propagate*, to allow the event and adapt the module to a new internal structure.

In this paper, we model ecosystems as graphs annotated with policies for responding to evolutionary events (Sec. 2) and we address the problem of identifying (a) what parts of the ecosystem are affected whenever we test a potential change and (b) how will the ecosystem look like once the implications of conflicting policies are resolved and the graph is appropriately rewritten (Sec. 3). Related work in ecosystem adaptation has provided us with techniques for view adaptation [1], [2], [3] that do not allow the definition of the policies for the adaptation of the ecosystem modules. Our previous work [4] has proposed algorithms for impact assessment with explicit policy annotation; however, to the best of our knowledge, there is no method that allows both the impact assessment and the rewriting of the ecosystem's modules along with correctness guarantees.

We implemented our method in a *what-if analysis* tool, Hecataeus[1] where all stakeholders can pre-assess the impact of possible modifications before actually performing them, in a way that is loosely coupled to the ecosystem's components. Our experimentation with ecosystems of different policies and sizes (Sec. 4) indicates that our method offers significant effort gains for the maintenance team of the ecosystem and, at the same time, scales gracefully.

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

2 Formal Background

Our modeling technique, extending [4], uniformly represents all the components of an ecosystem as a directed graph which we call the *Architecture Graph* of the ecosystem. Fig. 2 visually represents the internals of the modules of Fig. 1. To avoid overcrowding the figure, we omit different parts of the structure in different modules; the figure is self-explanatory on this.

Fig. 2. A subset of the graph structure for the University-DB Ecosystem

Modules. A module is a semantically high level construct of the ecosystem; specifically, the modules of the ecosystem are relations, views and queries. Every module defines a scope recursively: every module has one or more schemata in its scope (defined by part-of edges), with each schema including components (e.g., the attributes of a schema or the nodes of a semantics tree) linked to the schema also via part-of edges. In our model, all modules have a well defined scope, "fenced" by input and output schemata.

Relations. Each relation includes a node for the relation per se, a node for its (output) schema and a node for each for its attributes; all connected via the aforementioned part-of edges.

Queries. The graph representation of a Select - Project - Join - Group By (SPJG) query involves a new node representing the query, named *query node*, linked to the following schemata:

1. a set of *input schemata nodes* (one for every table appearing in the FROM clause). Each input schema includes the set of attributes that participate in the syntax of the query (i.e., SELECT, WHERE and GROUP BY clauses, etc.). Each input attribute is linked via a provider, *map-select* edge to the appropriate attribute of the respective provider module.
2. an *output schema node* comprising the set of attributes present in the SELECT clause. The output attributes are linked to the appropriate input attributes that populate them through *map-select* edges, directing from the output towards the input attributes.
3. 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).

We accommodate WHERE clauses in conjunctive normal form, where each atomic formula is expressed as: (i) Ω *op* constant, or (ii) Ω *op* Ω', or (iii) Ω *op* Q where Ω, Ω' are attributes of the underlying relations, Q is a nested query, and operator *op* belongs to the set $\{<, >, =, \leq, \geq, \neq, IN, EXISTS, ANY\}$). The entire WHERE clause is mapped to a tree, where (i) each atomic formula is mapped to a subtree with an operator node for *op* linked with *operand* edges pointing to the operand nodes of the formulae and (ii) nodes for the Boolean operators (AND, OR) connect with each other as well as with the operators of the atomic formulae via the respective operand edges. The GROUP BY part is mapped in the graph via (i) a node GB, to capture the set of attributes acting as the aggregators and (ii) 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*. The GB node is linked to the respective input attributes acting as aggregators with *group-by* edges, which are additionally tagged according to the order of the aggregators; we use an identifier i to represent the i-th aggregator. Moreover, for every aggregated attribute in the query's output schema, there exists a *map-select* edge directing from this attribute towards the aggregate function node as well as an edge from the function node towards the respective input attribute.

Views. 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 at the schema level with provider edges only among schemata (instead of attributes too).

Events. We organize the events that can be tested via our method in the following groups.

- *Events at relations.* A relation can withstand deletion and renaming of itself as well as addition, deletion and renaming of its attributes.
- *Events at views and queries.* A view can withstand the deletion and renaming of itself, the addition, deletion or renaming of its output attributes and the update of the view's semantics (i.e., the modification of the WHERE clause of the respective SQL query that defines the view).

Policies. As already mentioned, the policy of a node for responding to an incoming event can be one of the following: (a) PROPAGATE, which means that the node is willing to adapt in order to be compatible with the new structure and semantics of the ecosystem, or, (b) BLOCK, which means that the node wants to retain the previous structure and semantics. We can *assign policies* to all the nodes of the ecosystem via a language [5] that provides guarantees for the complete coverage of *all* the graph's nodes along with syntax conciseness and customizability. The main idea is the usage of rules of the form <receiver node> : on <event> then <policy>, both at the default level –e.g.,

VIEW.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;

and at the node-specific level (overriding defaults) –e.g.,

V_TR_OUT.SELF: on ADD_ATTRIBUTE then BLOCK;

3 Impact Assessment and Adaption of Ecosystems

The goal of our method is to assess the impact of a hypothetical event over an architecture graph annotated with policies and to adapt the graph to assume its new structure after the event has been propagated to all the affected modules. Before any event is tested, we topologically sort the modules of the architecture graph (always feasible as the summary graph is acyclic: relations have no cyclic dependencies and no query or view can have a cycle in their definition). This is performed once, in advance of any impact assessment. Then, in an on-line mode, we can perform what-if analysis for the impact of changes in two steps that involve: (i) the detection of the modules that are actually affected by the change and the identification of a status that characterizes their reaction to the event, and, (ii) the rewriting of the graph's modules to adapt to the applied change.

3.1 Detection of Affected Nodes and Status Determination

The assessment of the impact of an event to the ecosystem is a process that results in assigning every affected module with a status that characterizes its policy-driven response to the event. The task is reduced in (a) determining the affected modules in the correct order, and, (b) making them assume the appropriate status. Algorithm *Status Determination* (Fig. 3) details this process. In the following, we use the terms *node* and *module* interchangeably.

1. Whenever an 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 to ensure that a module is visited after *all* of its data providers have been visited. A visited node assesses the impact of the event internally (cf., "intra-module processing") and obtains a *status*, which can be one of the following: (a) BLOCK, meaning that the module is requesting that it remains structurally and semantically immune to the tested change and blocks the

Input: A topologically sorted architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, a global queue Q that facilitates the exchange of messages between modules.
Output: 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. Q=$\{original\ message\}$, $Affected\ Modules = \emptyset$;
2. For All $node \in \mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$
3. $\quad node.status\ =\ NO_STATUS$;
4. EndFor
5. While (size(Q) $>$ 0)
6. $\quad$ visit module ($node$) in head of Q;
7. $\quad$ insert $node$ in $Affected\ Modules$ list;
8. $\quad$ get all messages, $Messages$, that refer to $node$;
9. $\quad$ SetStatus($node$, $Messages$);
10. $\quad$ If ($node.status\ ==\ PROPAGATE$) Then
11. $\quad\quad$ insert $node.Consumers\ Messages$ to the Q;
12. EndWhile
13. Return $Affected\ Modules$;

Procedure SetStatus($Module$, $Messages$)
$Consumers\ Messages\ =\ \emptyset$;
For All $Message \in\ Messages$
  decide status of $Module$;
  put messages for $Module$'s consumers in $Consumers\ Messages$;
EndFor

Fig. 3. Algorithm Status determination

event (as its immunity obscures the event from its data consumers), (b) PROPAGATE, meaning that the modules concedes to adapt to the change and propagate the event to any subsequent data consumers, or, (c) retain a NO_STATUS status, already assigned by the topological sort, meaning that the module is not affected by the change.

2. 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 common *global message queue* (where messages are sorted by their target module's topological sorting identifier).
3. The process terminates whenever there are no more messages and no more modules to be visited.

Intra-module Processing. Whenever visited, a module starts by retrieving from the common queue *all* the messages (i.e., events) that concern it. It is possible that more than one message exist in the global queue for a module: e.g., with the deletion of an attribute that was used both 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. The processing of the messages is performed as follows:

1. 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.
2. Within each schema, the schema has to probe both itself and its contained nodes (attributes) for their reaction to the incoming event. At the end of this process, the schema assumes a status as previously discussed.
3. 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.

Theoretical Guarantees. Previous models of Architecture Graphs ([4]) allow queries and views to directly refer to the nodes representing the attributes of the involved relations. Due to the framing of modules within input and output schemata and the topological sorting, in [6] we have proved that the process (a) terminates and (b) correctly assigns statuses to modules.

3.2 Query and View Rewriting to Accommodate Change

Once the first step of the method, *Status Determination*, has been completed and each module has obtained a status, the problem of adaptation would intuitively seem simple: 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*. The example of Fig. 4 is illustrative of this case.

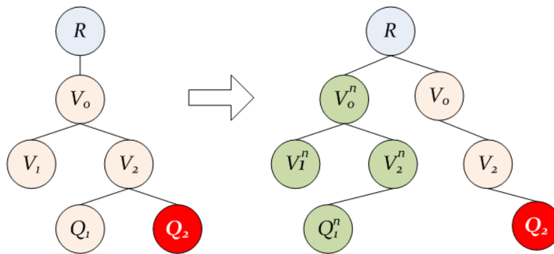

Fig. 4. Block rewriting example

In the example of Figure 4, we have a relation R and a view V_0 defined over the relation R. Two views (V_1 and V_2) use V_0 in order to get data. V_2 is further

used by two queries (Q_1 and Q_2). The database administrator wants to change V_0, in a way that all modules depending on V_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 Q_2 accept to adapt to the change, as they have a PROPAGATE policy annotation. Still, the vetoing Q_2 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 (V_0) to the blocking Q_2. As one can see in the figure, we now have two variants of V_0 and V_2: the new ones (named V_0^n and V_2^n) that are adapted to the new structure of V_0 – now named V_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 Q_2.

Input: An architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, a list of modules $Affected\ modules$, affected by the event, and the $Initial\ Event$ of the user.
Output: 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 that the user asked on the current version

1. For All $Module \in Affected\ modules$
2. If($Module.status == BLOCK$) Then
3. CheckModule($Module$, $Affected\ modules$, $Initial\ Event$);
4. mark $Module$ not to change; //Blockers do not change
5. EndFor

Procedure CheckModule($Module$, $Affected\ modules$, $Initial\ Event$)
If($Module$ has been marked) Then return; //Notified by previous block path
If ($Initial\ Event == ADD_ATTRIBUTE$)
Then mark $Module$ to apply change on current version; //Blockers ignore provider addition
Else mark $Module$ to keep current version as is and apply the change on a clone;
For All $Module\ provider \in Affected\ modules$ feeding $Module$
CheckModule($Module\ provider$, $Affected\ modules$, $Initial\ Event$); //Notify path
EndFor

Fig. 5. Algorithm PATH CHECK

The crux of the problem is as follows: if a module has PROPAGATE status and none of its consumers (including both its immediate and its transitive consumers) raises a BLOCK veto, 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, then *the entire 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 serve the correct definition of the vetoing module.

To correctly serve the above purpose, the adaptation process is split in two steps. The first of them, *Path Check*, works from the consumers towards the providers in order to determine the number of variants (old and new) for each

module. Whenever the algorithm visits a module, if its status is BLOCK, it starts a reverse traversal of the nodes, starting from the blocker module towards the module that initialized the flow and marks each module in that path (a) to keep its present form and (b) prepare for a cloned version (identical copy) where the rewriting will take place. The only exception to this rewriting is when the module of the initial message is a relation module and the event is an attribute deletion, in which case a BLOCK signifies a veto for the adaptation of the relation.

Input: A list of modules $Affected\ modules$, knowing the number of versions they have to retain, initial messages of $Affected\ modules$
Output: Architecture graph after the implementation of the change the user asked

1. If(any of $Affected\ modules$ has status BLOCK) Then
2. If(initial message started from Relation module type AND event == DELETE_ATTRIBUTE) Then Return;
3. Else
4. For All ($Module \in\ Affected\ modules$)
5. If($Module$ needs only new version) Then
6. proceed with rewriting of $Module$;
7. connect $Module$ to new providers; //new version goes to new path
8. Else
9. clone $Module$; //clone module, to keep both versions
10. connect cloned $Module$ to new providers; //clone is the new version
11. proceed with rewriting of cloned $Module$;
12. EndFor
13. Else
14. For All $Module \in\ Affected\ modules$
15. proceed with rewriting of $Module$ //no blocker node
16. EndFor

Fig. 6. Algorithm Graph Rewrite

Finally, all nodes that have to be rewritten are getting their new definition according to their incoming events. Unfortunately, this step cannot be blended with *Path Check* straightforwardly: *Path Check* operates from the end of the graph backwards, to highlight cases of multiple variants; rewriting however, has to work from the beginning towards the end of the graph in order to correctly propagate information concerning the rewrite (e.g., the names of affected attributes, new semantics, etc.). So, the final part of the method, *Graph Rewrite*, visits each module and rewrites the module as follows:

– If the module must retain only the new version, once we have performed the needed change, we connect it correctly to the providers it should have.
– If the module needs both the old and the new versions, we make a clone of the module to our graph, perform the needed change over the cloned module and connect it correctly to the providers it should have.
– If the module retains only the old version, we do not perform any change.

4 Experiments

We assessed our method for its usefulness and scalability with varying graph configurations and policies; in this section, we report our findings.

Experimental Setup. We have employed TPC-DS, version 1.1.0 [7] as our experimental testbed. 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: (a) the *Web*, or, (b) a *Catalog*, or, (c) directly at the *Store*. Since the Hecataeus' parser could not support all the advanced SQL constructs of TPC-DS, we employed several auxiliary views and slight query modifications.

Graphs and Events. To test the effect of graph size to our method's efficiency, we have created 3 graphs with gradually decreasing number of query modules: (a) a large ecosystem, *WCS*, with queries using all the available fact tables, (b) an ecosystem *CS*, where the queries to WEB_SALES have been removed, and (c) an ecosystem *S*, with queries using only the STORE_SALES fact table. The event workload consists of 51 events simulating a real case study of the Greek public sector. See Fig. 7 for an analysis of the module sizes within each scenario and the workload (listing the percentage of each event type as *pct*).

Policies. We have annotated the graphs with policies, in order to allow the management of evolution events. We have used two "profiles": (a) *MixtureDBA*, consisting of 20% of the relation modules annotated with BLOCK policy and (b) *MixtureAD*, consisting of 15% of the query modules annotated with BLOCK policy. The first profile corresponds to a developer-friendly DBA that agrees to prevent changes already within the database. The second profile tests an environment where the application developer is allowed to register veto's for the evolution of specific applications (here: specific queries). We have taken care to pick queries that span several relations of the database.

	Graph size		
	S	CS	WCS
Queries	27	68	89
Views	25	48	95
Relations	25	25	25
Sum	**77**	**141**	**218**

Event type	*pct*
Attribute Add	37.3%
Attribute Rename	43.2%
Attribute Del	13.7%
Relation Rename	1.9%
View alter semantics	3.9%

Fig. 7. Experimental configuration for the TPC-DS ecosystem

Experimental Protocol. 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 – i.e., each event is applied over the graph that resulted from the application of the previous event. We have performed our experiments with hot cache. For each event we measure the elapsed time for each of the three algorithms, along with the number of affected, cloned and adapted modules. All the experiments have been performed in a typical PC with an Intel Quad core CPU at 2.66GHz and 1.9GB main memory.

Effectiveness. How useful is our method for the application developers and the DBA's? We can assess the effort gain of a developer using the highlighting of affected modules of Hecataeus compared to the situation where he would have to perform all checks by hand as the *fraction of Affected Modules of the ecosystem*. This gain, expressed via the $\%AM$ metric 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 1.

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

	%AM – Mixture AD			%AM – Mixture DBA		
	S	CS	WCS	S	CS	WCS
min	21%	35%	30%	60%	78%	84%
avg	**89%**	**91%**	**92%**	**97%**	**96%**	**97%**
max	100%	100%	100%	100%	100%	100%

Fig. 8. Effectiveness assessment as fraction of affected modules (%AM)

The results depicted in Fig. 8 demonstrate that the effort gains compared to the absence of our method are significant, as, on average, the effort is around 90% in the case of the AD mixture and 97% in the case of the DBA mixture. As the graph size increases, the benefits from the highlighting of affected modules that our method offers, increase too. Observe that in the case of the DBA case, where the flooding of events is restricted early enough at the database's relations, the minimum benefit in all 51 events ranges between 60% - 84%.

Effect of Policy to the Execution Time. In the case of *Mixture DBA* we follow an aggressive blocking policy that stops the events early enough, at the relations, before they start being propagated in the ecosystem. On the other hand, in the case of *Mixture AD*, we follow a more conservative annotation approach, where the developer can assign blocker policies only to some module parts that he authors. In the latter case, it is clear that the events are propagated to larger parts of the ecosystem resulting in higher numbers of affected and rewritten nodes. If one compares the execution time of the three cases of the AD mixture in Fig. 9 with the execution time of the three cases of the DBA mixture the difference is in the area of one order of magnitude. It is however interesting to note the internal differences: the status determination time is scaled up with a factor of two; the rewriting time, however is scaled up by a factor of 10, 20 and 30 for the small, medium and large graph respectively!

Another interesting finding concerns the **internal breakdown of the execution time** in each case. A common pattern is that *path check is executed very efficiently*: in all cases it stays within 2% of the total time (thus practically invisible in the graphic). In the case of the AD mixture, the analogy between the status determination and the graph rewriting part starts from a 24% - 74% for

the small graph and ends to a 7% - 93% for the large graph. In other words, *as the events are allowed to flow within the ecosystem, the amount of rewriting increases with the size of the graph*; in all cases, it dominates the overall execution time. This is due to the fact that rewriting involves memory management (module cloning, internal node additions, etc) that costs much more than the simple checks performed by *Status Determination*. In the case of the DBA mixture, however, where the events are quickly blocked, the times are not only significantly smaller, but also equi-balanced: 57% - 42% for the small graph (*Status Determination* costs more in this case) and 49% - 50% for the two other graphs. Again, this is due to the fact that the rewriting actions are the time consuming ones and therefore, their reduction significantly reduces the execution time too.

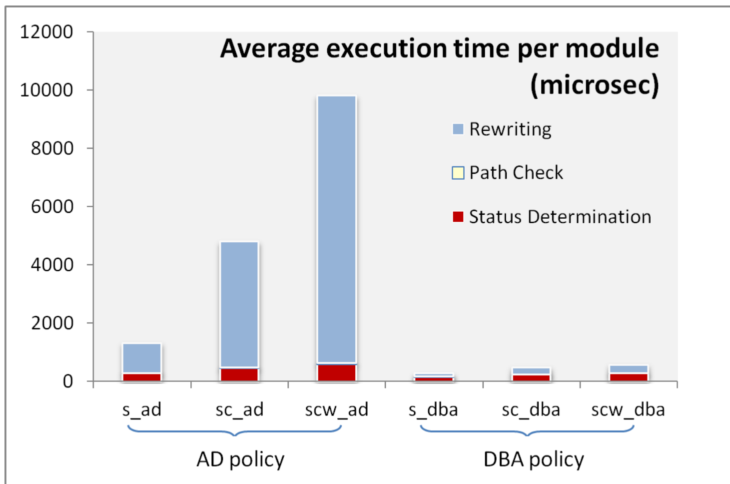

Fig. 9. Efficiency assessment for different policies, graph sizes and phases

Effect of Graph Size to the Execution Time. To assess the impact of graph size to the execution time one has to compare the three different graphs to one another within each policy. In the case of the AD mixture, where the rewriting dominates the execution time, there is a linear increase of both the rewriting and the execution time with the graph size. On the contrary, the rate of increase drops in the case of the DBA mixture: when the events are blocked early, the size of the graph plays less role to the execution time.

Overall, the main lesson learned from these observations is that the annotation of few database relations significantly restricts the rewriting time (and consequently the overall execution time) when compared to the case of annotating modules external to the database. In case the rewriting is not constrained early enough, then the execution cost grows linearly with the size of the ecosystem.

5 Related Work

For an overview of the vast amount of work in the area of evolution, we refer the interested reader to an excellent, recent survey [8]. We also refer the interested reader to [9] for a survey of efforts towards bidirectional transformations. Here, we scope our discussion to works that pertain to the adaptation of data-intensive ecosystems.

Data-Intensive Ecosystems' Evolution. Research activity around data-intensive ecosystems has been developed around two tools, Hecataeus and Prism. Hecataeus [4] models ecosystems as Architecture Graphs and allows the definition of policies, the impact assessment of potential changes and the computation of graph-theoretic properties as metrics for the vulnerability of the graph's nodes to change. The impact assessment mechanism was first introduced in [4] and subsequently modified in [6]. PRISM++ [10] 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.

View/Schema Mapping Rewriting. Nica et al., [1] make legal rewritings of views affected by changes and they primarily deal with the case of relation deletion by finding valid replacements for the affected (deleted) components via a meta-knowledge base (MKB) that keeps meta-information about attributes and their join equivalence attributes on other tables in the form of a hyper-graph. Gupta et al., [2] 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 compute the new view contents incrementally and avoid their full re-computation. Velegrakis, et al., [3], deal with the maintenance of a set of mappings in an environment where source and target schemata are integrated under schema mappings implemented via SPJ queries. Cleve et al., [11] introduce mappings among the applications and a conceptual representation of the database, again mapped to the database logical model; when the database changes, the mappings allow to highlight impacted areas in the source code.

Comparison to Existing Approaches. As already mentioned, the annotation of the ecosystem with policies imposes the new problem of maintaining different replicas of view definitions for different consumers; to the best of our knowledge, this is the first time that this problem is handled in a systematic way. Interestingly, although the existing approaches make no explicit treatment of policies, they differ in the implicit assumptions they make. Nica et al., operating mainly over virtual views [1], actually block the flooding of a deletion event by trying to compensate the deletion with equivalent expressions. At the same time, they do not handle additions or renamings. Velegrakis et al. [3] move towards the same goal but only for SPJ queries. On the other hand, Gupta et al., [2], working with materialized views, are focused to adapting the contents of the views, in a propagate-all fashion. A problem coming with a propagate-all policy is that the events might affect the semantical part of the views/queries (WHERE clause)

without any notification to the involved users (observe that the problem scales up with multiple layers of views defined over other views).

Compared to previous editions of Hecataeus [4], this work reports on the first implementation of a status determination mechanism with correctness guarantees. The management of rewritings via the path checking to handle conflicting policies and the adaptation to accommodate change are completely novel.

6 Conclusions and Future Work

In this paper we have addressed the problem of adapting a data-intensive ecosystem in the presence of policies that regulate the flow of evolution events. Our method allows (a) the management of alternative variants of views and queries and (b) the rewriting of the ecosystem's affected modules in a way that respects the policy annotations and the correctness of the rewriting (even in the presence of policy conflicts). Our experiments confirm that the adaptation is performed efficiently as the size and complexity of the ecosystem grow. Future work can address the assessment of complicated events, the visualization of the ecosystem and the automatic suggestion of policies.

Acknowledgments. This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

1. Nica, A., Lee, A.J., Rundensteiner, E.A.: The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In: Schek, H.-J., Saltor, F., Ramos, I., Alonso, G. (eds.) EDBT 1998. LNCS, vol. 1377, pp. 359–373. Springer, Heidelberg (1998)
2. Gupta, A., Mumick, I.S., Rao, J., Ross, K.A.: Adapting materialized views after redefinitions: techniques and a performance study. Information Systems 26(5), 323–362 (2001)
3. Velegrakis, Y., Miller, R.J., Popa, L.: Preserving mapping consistency under schema changes. VLDB Journal 13(3), 274–293 (2004)
4. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: Policy-Regulated Management of ETL Evolution. J. Data Semantics 13, 147–177 (2009)
5. Manousis, P.: Database evolution and maintenance of dependent applications via query rewriting. MSc. Thesis, Dept. of Computer Science, Univ. Ioannina (2013), `http://www.cs.uoi.gr/~pmanousi/publications.html`
6. Papastefanatos, G., Vassiliadis, P., Simitsis, A.: Propagating evolution events in data-centric software artifacts. In: ICDE Workshops, pp. 162–167 (2011)
7. Transaction Processing Performance Council: The New Decision Support Benchmark Standard (2012), `http://www.tpc.org/tpcds/default.asp`

8. Hartung, M., Terwilliger, J.F., Rahm, E.: Recent Advances in Schema and Ontology Evolution. In: Schema Matching and Mapping, pp. 149–190. Springer (2011)
9. Terwilliger, J.F., Cleve, A., Curino, C.: How clean is your sandbox? - towards a unified theoretical framework for incremental bidirectional transformations. In: 5th Intl. Conf. Theory and Practice of Model Transformations (ICMT), Prague, Czech Rep., pp. 1–23 (2012)
10. Curino, C., Moon, H.J., Deutsch, A., Zaniolo, C.: Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. PVLDB 4(2), 117–128 (2010)
11. Cleve, A., Brogneaux, A.-F., Hainaut, J.-L.: A conceptual approach to database applications evolution. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 132–145. Springer, Heidelberg (2010)