HECATAEUS: Regulating Schema Evolution

George Papastefanatos[#1], Panos Vassiliadis[*2], Alkis Simitsis[⋆3], Yannis Vassiliou[#4]

[#] *National Technical University of Athens, Greece*
[1]gpapas@dblab.ntua.gr [4]yv@cs.ntua.gr
[*] *University of Ioannina, Greece*
[2]pvassil@cs.uoi.gr
[⋆] *HP Labs, USA*
[3]alkis@hp.com

Abstract— **HECATAEUS is an open-source software tool for enabling impact prediction, what-if analysis, and regulation of relational database schema evolution. We follow a graph theoretic approach and represent database schemas and database constructs, like queries and views, as graphs. Our tool enables the user to create hypothetical evolution events and examine their impact over the overall graph before these are actually enforced on it. It also allows definition of rules for regulating the impact of evolution via (a) default values for all the nodes of the graph and (b) simple annotations for nodes deviating from the default behavior. Finally, HECATAEUS includes a metric suite for evaluating the impact of evolution events and detecting crucial and vulnerable parts of the system.**

I. MOTIVATION

Database-centric systems are continuously evolving environments, where design constructs are added, removed or updated due to changes to system specifications and initial requirements. Hence, database schema evolution imposes many challenges to all development phases of a data-centric system. Apart from their schema, database centric environments comprise a large number of applications and data stores related to such systems, which are also affected by evolution events. For example, any change in the database schema may affect queries embedded in procedures, views, software modules, complex workflows, and so on. In doing so, queries and data entry forms can be invalidated and application programs may crash.

In our research, we have identified three fundamental needs of the developer, administrator, and designer of a database.

A developer having constructed a large set of modules, reports, scripts, and so on, is affected by changes in the database schema. Each change possibly makes the developer's artifacts syntactically inconsistent (in which case they crash) or semantically dubious (in which case they return the wrong results to the users). The developers would enjoy a facility that *predicts and evaluates the effect of a schema evolution event and highlights places where the code must be maintained*.

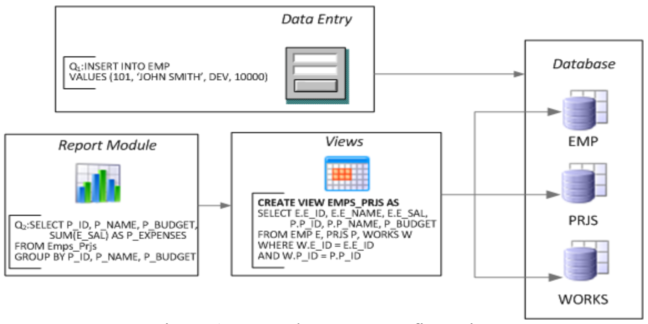
It is possible that predicting the effect of a change in a database may cover a large part of the deployed code, whereas in practice this could be constrained via an 'API-like' set of views. Interestingly, a view can act as an airbag to evolution events, masking them from the developer and transferring the need for compensation to the database administrator. In this case, the administrator would need *a means (i.e., an effect prediction system) to control the flooding of the event's impact to the affected constructs*.

At the same time, the designer should consider evolution during the database design. The fundamental question need to be addressed is "is design A better than design B when it comes to the effort that will be required for maintaining them in the presence of evolution events?" The designer can highly benefit from *a set of objective metrics that report the design quality with respect to the vulnerability to evolution events*.

Currently, little or even no support is provided from current RDBMSs for analysing the impact and furthermore, adjusting semantic and syntactical inconsistencies as results of evolution events. To the best of our knowledge, problems related to evolution are handled manually by administrators and developers. On the other hand, research efforts have focused mainly on the adaptation of internal database objects to schema changes, without giving proper attention to existing queries and views and their role as integral parts of the environment.

To handle evolution events, we have developed novel methods to this problem [1, 2], which are implemented in our tool named HECATAEUS. HECATAEUS provides the user with the means for executing evolution scenarios and predicting the impact of a potential change to the system. In a nutshell, first we extract from a RDBMS relational schemas along with dependent constructs like queries and views and then, we represent them as directed graphs. Working with such graphs, HECATAEUS enables the user to create hypothetical evolution events and examine their impact over a graph. It also allows the definition of rules for regulating the evolution impact to the system and automates its adaptation to evolution events. In addition, Hecataeus supports an extensible suite of design metrics, which can be used for detecting crucial and vulnerable parts of the system regarding potential evolution events.

Motivating Example. Next, we present HECATAEUS using the example configuration of Figure 1, which involves 3 relations holding data for employees and the projects they work for. Assume that in the application layer, data entry forms are used for inserting and updating records in the database, while a report module is also used for analytical purposes. This module interacts with the database through a view layer. The view layer comprises a view, namely `Emps_Prjs`, which relates employees to their projects. On top of this view, the report module involves an aggregate query that calculates the monthly expenses of each project by summing up the salaries of all employees working for it and compares these salaries with the project budget. In the data entry forms, an `INSERT` statement adds records in `EMP` relation.



```
Q1:INSERT INTO EMP
VALUES (101, 'JOHN SMITH', DEV, 10000)
```

```
Q2:SELECT P_ID, P_NAME, P_BUDGET,
        SUM(E_SAL) AS P_EXPENSES
FROM Emps_Prjs
GROUP BY P_ID, P_NAME, P_BUDGET
```

```
CREATE VIEW EMPS_PRJS AS
SELECT E.E_ID, E.E_NAME, E.E_SAL,
        P.P_ID, P.P_NAME, P_BUDGET
FROM EMP E, PRJS P, WORKS W
WHERE W.E_ID = E.E_ID
AND W.P_ID = P.P_ID
```

Figure 1: Example system configuration

Which parts of the system are affected and how? Assume that due to an evolving business requirement, a new attribute, e.g., `MOB_PHONE`, has to be added in the `EMP` relation. Should this change be propagated to the view and/or the query involved in the data entry form? Is actually the query affected by such an event? Although attribute deletion can be handled easily since queries become syntactically incorrect, the addition of information is deferred to a decision of the designer.

Can we predefine the reaction of the system to potential changes and regulate the impact of evolution? The situation gets complicated when the attributes added involve primary keys. Assume we add two attributes to `WORK` relation, namely `StartDate` and `EndDate`, characterizing the period over which an employee has worked on a project. In this case, the uncertainty on the correctness of the view definition is increased. Do we request all employees ever having worked in a project or only the ones currently involved in some project? Similar considerations arise in the case where the `WHERE` clause of the view is modified. Assume that a field `STATUS` is added to all projects and the view definition is modified by incorporating the extra selection `STATUS='Active'`. Can we still use the view in order to answer the query or not? The answer is not obvious, since it depends on whether the query employed by the analysts, uses the view simply as a macro (for avoiding the extra coding effort) or whether the query is supposed to work on the view, independently of what the view definition is.

Can we measure the quality of a design with objective quality metrics, keeping evolution (apart from performance and space considerations) in mind? Additionally, it is useful to identify the parts of the overall configuration that are most prone to be affected by potential schema changes at the underlying relations. For example, the `Emps_Prjs` view can be considered as a sensitive part of the system, since it depends on all three relations and a report depends on it. Is it safer to opt for a design without the sensitive view or is it better to keep it and use it as a change buffering aid for the developer?

Clearly, in real-world settings things are more complicated. We support a powerful language [3], graph visualization techniques [2], and an extensible metric suite [1] that enable us to automatically predict and handle evolution events and propagate policies to the full extent of large-scale scenarios, without imposing significant overhead to the designer. We have tested our tool in real-world problems and applications and the results of its use have been proven really helpful for designers and very encouraging for us. Next, we briefly present the main functionality provided by HECATAEUS.

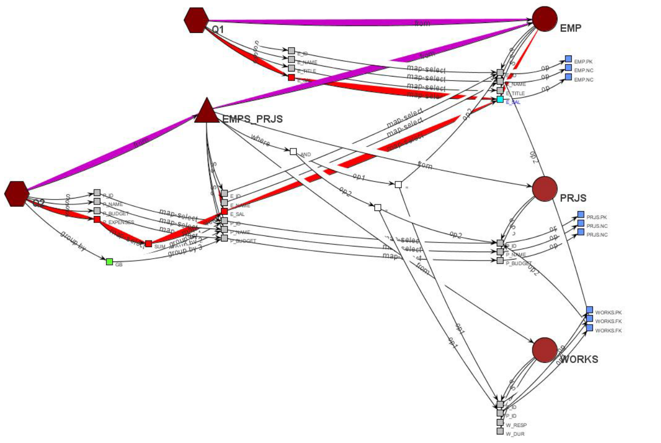


Figure 2: Highlighting the impact of deleting attribute *EMP.E_SAL*

II. THEORETICAL UNDERPINNINGS

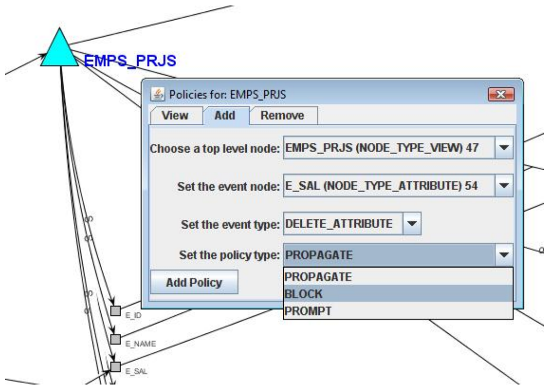
We use a graph model, termed *Evolution Graph*, that models in a coherent and uniform way: (a) internal structural elements of databases, such as relations, views, triggers; and (b) external components accessing databases, such as queries extracted from procedures, and object modules along with their significant properties (e.g., attributes and conditions).

When an evolution event is tested, the impact of the event is automatically computed throughout the subset of the affected graph via a messaging mechanism between the nodes. We also provide mechanisms to regulate the flooding of the messages travelling across the evolution graph and block the propagation of events in places that should prevent further flooding [3]. Assume that query node `Q2` is annotated with a policy that blocks the inclusion of added attributes in `Emps_Prjs` in the query's select clause:

```
Q2: SELECT EP.Emp#, EP.Name FROM emps_prjs EP
       WHERE EP.PRJNAME = 'Olympic Games'
       ON Add Attribute TO emps_prjs THEN block
```

When the administrators examine what will happen if an attribute is added to the view, the event will not affect the query `Q2`, since it is immune to this kind of change. If `Q2` was a view too, queries using it would never be affected by the change, either. Human interaction is minimized by expressing policies via (a) *default values* for all parts of the graph and (b) *SQL extensions* for deviations from defaults (as above). Our annotations cover a broad range of additions and deletions to the graph concerning attributes, constraints, semantics hidden in the `WHERE` clause of views, and other similar events [3].

Finally, we employ graph and information theoretic properties of the evolution graph and establish a set of measurements for evaluating the design quality of database centric environments with respect to its ability to sustain evolution operations [2]. Specifically, we have provided evidence that metrics like the degrees (in, out, and total) of a node, the transitive degrees of a node (i.e., the extent to which other nodes transitively depend upon it), and the degrees of a summarized variant of a module (e.g., a view) can be good predictors of the sensitivity of a database construct with respect to its evolution. We also consider a module's entropy that simulates the extent to which the vulnerability of a node is surprising.



Policies for: EMPS_PRJS

View | Add | Remove

Choose a top level node: EMPS_PRJS (NODE_TYPE_VIEW) 47

Set the event node: E_SAL (NODE_TYPE_ATTRIBUTE) 54

Set the event type: DELETE_ATTRIBUTE

Set the policy type: PROPAGATE

PROPAGATE
BLOCK
PROMPT

Add Policy

Figure 3: Definition of policy Block on view *EMPS_PRJS*

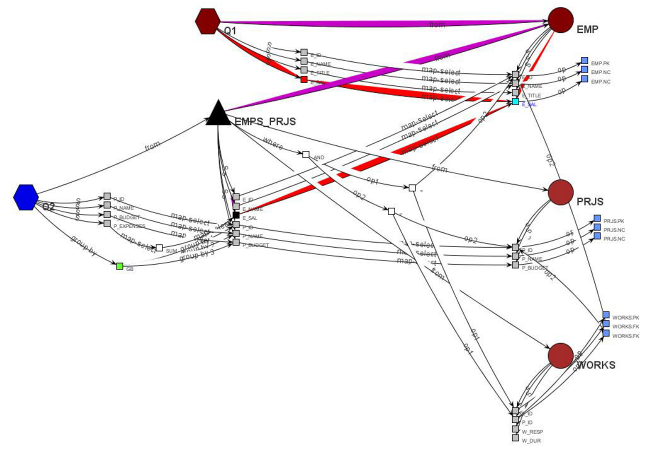


Figure 4: Propagating deletion of *EMP_E_SAL* is blocked by *EMPS_PRJS*

III. Demonstration

This section briefly presents HECATAEUS functionality using references to Figure 1.

Representation of schema constructs as directed graphs. HECATAEUS input involves SQL files containing database schema definitions, queries, and views definitions. Also, it can automatically retrieve metadata (i.e., tables and views definition) from various RDBMs. Schema definitions and SQL code are parsed for creating the *Evolution Graph*, which may represent database schemata at the most granular level. Each node represents a database object (e.g., a relation or an attribute), and each edge represents a relationship between these objects. Figure 2 shows an instance of our example evolution graph.

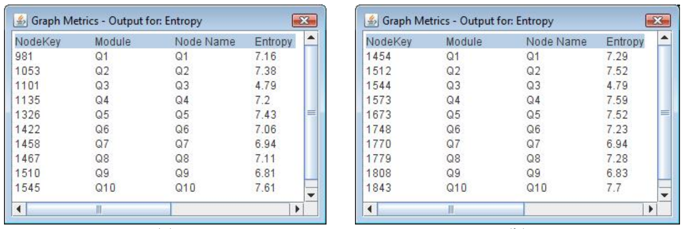
Users can interactively work on the graph. Specifically, they may apply layout algorithms to transform the way the graph is displayed, and find, hide, add or remove graph constructs or even change their name and type. Additionally, to tackle the problems of limited screen real-estate and effective graph navigation, we offer capabilities for zooming on parts of the graph, isolating modules and creating subgraphs, and displaying the graph at various abstraction levels (i.e., only top level nodes for relations, views and queries).

Impact prediction of evolution changes. HECATAEUS provides the user with the ability to simulate the occurrence of evolution processes on the database schema and experiment with the impact that these processes have on the system before these are actually enforced on it. The construction of *hypothetical evolution scenarios* is one of the major features of the tool. This has been proven extremely handy in real-world projects during both their design and deployment phases. Users can define events on nodes of the graph corresponding to evolution changes on parts of the database schema, such as addition of attributes, deletion of relations, and so on. The triggering of a hypothetical event highlights all nodes that are semantically or syntactically affected by this event. That is, nodes are assigned with a status that describes their reaction to this event and they are coloured accordingly. Figure 2 shows the impact of a hypothetical deletion of attribute E_SAL from relation EMP. The nodes affected by this deletion are coloured according to the impact that the event has on them (e.g., the red nodes are not defined in the absence of EMP.E_SAL attribute whereas the brown nodes are syntactically affected).

Regulating Schema Evolution. Apart from defining and triggering evolution events, HECATAEUS enables the definition of rules, also called *policies*, on the nodes of the graph. The crux of our approach involves annotating the graph constructs (as relations, attributes, and conditions) that sustain changes (as addition, deletion, and modification) with policies dictating the way they will regulate the change. Three policies are defined: (a) *propagate the change*: the graph must be reshaped to adjust to the new semantics incurred by the event; (b) *block the change*: the old semantics of the graph are retained and the event is blocked or constrained through some rewriting that preserves the old semantics; and (c) *prompt*: the user should decide what will eventually happen. The combination of events with policies triggers the execution of actions that either block the event or highlight properly the graph to adapt to the proposed change. Hence, the adaptation of a node to an evolution event and furthermore, the propagation of the event towards the rest of the graph are dictated by the rule defined on the node. Again, default values and policies drastically reduce the human intervention required in this task.

Figure 3 shows the annotation of EMPS_PRJS view with a policy for blocking the deletion of attribute E_SAL from its schema and for propagating the deletion to any dependent constructs. The highlighting of the graph in the presence of policies that act as evolution regulators is shown in Figure 4. The deletion of EMP.E_SAL is blocked (shown as black coloured) at the view level and query Q_2 is immune to this change.

Evaluation of Design Metrics. HECATAEUS offers a set of design metrics, which act as predictors for the vulnerability of graph constructs to future evolution changes. They also assess the quality of design alternatives with respect to their ability to seamlessly sustain evolution changes. Figure 5 reports on the results of the entropy metric for all TPC-DS queries targeting two alternative schemas regarding the Web-Sales part of TPC-DS benchmark [7]. Figure 5a concerns the original schema, and Figure 5b concerns an alternative design having an intermediate view layer defined on top of the TPC-DS schema, placed between queries and the underlying relations. Potential events occurring at the view layer increase the overall vulnerability of the queries to future changes, and this fact results in higher entropy values for the alternative design.



Graph Metrics - Output for: Entropy

NodeKey	Module	Node Name	Entropy
981	Q1	Q1	7.16
1053	Q2	Q2	7.38
1101	Q3	Q3	4.79
1135	Q4	Q4	7.2
1326	Q5	Q5	7.43
1422	Q6	Q6	7.06
1458	Q7	Q7	6.94
1467	Q8	Q8	7.11
1510	Q9	Q9	6.81
1545	Q10	Q10	7.61

(a)

Graph Metrics - Output for: Entropy

NodeKey	Module	Node Name	Entropy
1454	Q1	Q1	7.29
1512	Q2	Q2	7.52
1544	Q3	Q3	4.79
1573	Q4	Q4	7.59
1673	Q5	Q5	7.52
1748	Q6	Q6	7.23
1770	Q7	Q7	6.94
1779	Q8	Q8	7.28
1808	Q9	Q9	6.83
1843	Q10	Q10	7.7

(b)

Figure 5: Output of *Entropy* metric for all queries targeting (a) TPC-DS, (b) TPC-DS variant with views

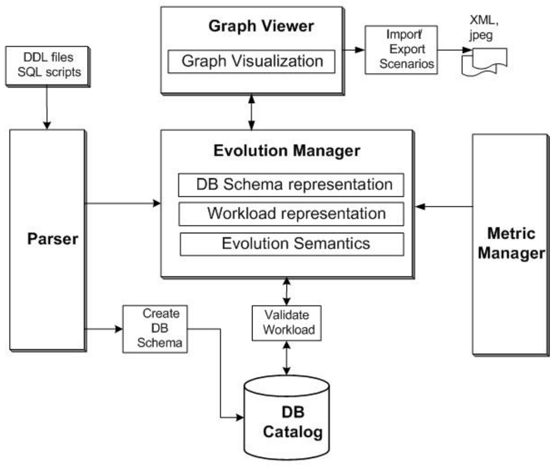


Figure 6: System Architecture

IV. Architecture

HECATAEUS architecture comprises five main components: the Parser, the Evolution Manager, the Graph Viewer, the Metric Manager, and the Catalog (Figure 6).

The *Parser* parses the input files (i.e., DDL and workload definitions) and sends each command to the database Catalog and to the Evolution Manager. It also parses language extensions for policy definitions on the graph.

The *Catalog* keeps track of the relation schemata and validates the syntax of the parsed workload (i.e., activity definitions, queries, views), before Evolution Manager models them.

The *Evolution Manager* represents the underlying database schemata and the parsed queries in the proposed graph model. It holds the semantics of nodes and edges, and assigns nodes and edges to their respective classes. It communicates with the Catalog and the Parser and constructs the node and edge objects for each class of nodes and edges (i.e., relation nodes, query nodes, etc.). It retains all evolution semantics for each graph construct (e.g., events, policies) and methods for performing evolution scenarios. Finally, it contains methods for transforming the database graph from/to an XML format.

The *Metric Manager* maintains the metrics definition and applies them to the graph. Each metric is implemented as a separate function. Therefore, our design and implementation are extensible in that they allow the smooth incorporation of additional design metrics.

The *Graph Viewer* is responsible for the visualization of the graph and the interaction with the user. It communicates with the Evolution Manager, which holds all evolution semantics

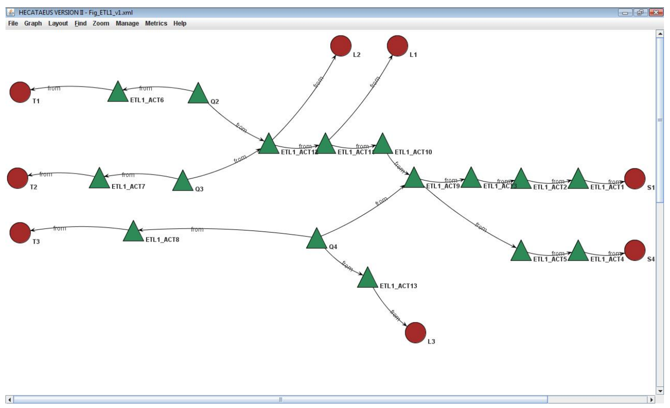


Figure 7: Example workflow representing the population of a data warehouse

and methods. In addition, evolution scenarios can be imported or exported to XML format and saved to image formats (as jpeg). Finally, we provide several graph manipulation operations (e.g., zooming in/out) to improve the readability and usability of our approach.

HECATAEUS is an open source project implemented in Java [4]. The parser and database engine have been built on top of HSQLDB, an open source SQL relational database engine [5]. For the graph visualization, we have used the Java Universal Network/Graph Framework (JUNG), a software library for the management of graphs and networks [6]. JDBC connections are used for communicating with the most popular RDBMSs.

V. Evaluation

We have evaluated the effectiveness and scalability of HECATAEUS in several real-world applications. A particular case study concerned the reverse engineering of integration workflows maintaining farming and agricultural statistics, which were extracted from an application of the Greek public domain. The integration scenarios comprised a total number of 59 jobs which extract information out of a set of 7 source schemata and load it to a data warehouse. Figure 7 shows a high level representation of a particular integration workflow involving 16 jobs. In that application, we used HECATAEUS for handling a total number of 374 evolution events occurred at the source schemata over a period of 6 months. The in-situ demo will demonstrate abstracted cases from the above application and also, representative scenarios based on the TPC-DS benchmark, which will show how our tool can conveniently accommodate large-scale graphs.

References

[1] G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou. *Design Metrics for Data Warehouse Evolution*. In ER, 2008.

[2] 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.

[3] G. Papastefanatos, P. Vassiliadis, A. Simitsis, K. Aggistalis, F. Pechlivani, Y. Vassiliou. *Language Extensions for the Automation of Database Schema Evolution*. In ICEIS, 2008.

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

[5] HSQL Database Engine: http://hsqldb.org

[6] Java Universal Network/Graph Framework (JUNG): http://jung.sourceforge.net/index.html

[7] TPC-DS Benchmark. http://www.tpc.org/tpcds/default.asp