Design Metrics for Data Warehouse Evolution

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

[1] National Technical University of Athens, Athens, Hellas
{gpapas,yv}@dbnet.ece.ntua.gr
[2] University of Ioannina, Ioannina, Hellas
pvassil@cs.uoi.gr
[3] Stanford University and HP Labs, California, USA
alkis@{db.stanford.edu,hp.com}

Abstract. During data warehouse design, the designer frequently encounters the problem of choosing among different alternatives for the same design construct. The behavior of the chosen design in the presence of evolution events is an important parameter for this choice. This paper proposes metrics to assess the quality of the warehouse design from the viewpoint of evolution. We employ a graph-based model to uniformly abstract relations and software modules, like queries, views, reports, and ETL activities. We annotate the warehouse graph with policies for the management of evolution events. The proposed metrics are based on graph-theoretic properties of the warehouse graph to assess the sensitivity of the graph to a set of possible events. We evaluate our metrics with experiments over alternative configurations of the same warehouse schema.

1 Introduction

How good is the design of a data warehouse? What makes the design of a data warehouse good or bad? Typically, such questions are answered by a set of empirical rules, such as 'are your dimensions aligned?', 'is the warehouse following a typical design pattern, such as star or snowflake?', 'are the partitions and indexes of the warehouse built appropriately?', and so on. All these recipes are based on practical observations of the past, as well as rules of thumb that have been established by expert practitioners and although valuable, they simply transfer the lessons learned the hard way in the "craft" of data warehouse design.

At the same time, the scientific community is not in possession of a fundamentally established theory for the evaluation of the quality of a data warehouse. So far, the researchers have dealt with metrics that evaluate the design quality of the database schema with respect to high level goals, such as completeness, understandability, etc. both at the conceptual [16] and the logical level [3, 9]. Although structural properties of the database or the warehouse (e.g., number of dimensions or foreign keys) are considered, the employed approaches restrict themselves to constructs internal to the database without taking into account the incorporation of constructs surrounding the database into their models, nor the fact that a software construct, and especially an information system, evolves over time. Since software maintenance makes up for at least 50% of all resources spent in a project, maintainability is an important factor for

Q. Li et al. (Eds.): ER 2008, LNCS 5231, pp. 440–454, 2008.
© Springer-Verlag Berlin Heidelberg 2008

the determination of the quality of a design. The problem is quite hard, since changes in the schema of a database-centric system (and thus, a data warehouse) affect both its internals but also, the surrounding deployed applications. Thus, the minimal interdependence of these software modules results in higher tolerance to subsequent changes and should be measured with a principled theory. Related work for view redefinition [5, 8, 10] and data warehouse evolution [2, 4, 6, 7] has provided rewriting techniques and theoretical cost models; yet, a well founded model that captures all the environment of a warehouse and objectively assess its vulnerability to changes is missing.

In this paper, we propose a set of metrics with two major characteristics. Firstly, they act as predictors for the vulnerability of a software module of a data warehouse (either internal, e.g., a dimension table, or external, e.g., an aggregated measure in a user's report) to future changes to the structure of the warehouse. Secondly, they facilitate the assessment of the quality of alternative designs of the warehouse with a particular viewpoint on the evolution of the data warehouse.

To achieve the abovementioned goal, we base our approach on two pillars.

First, we model the whole environment of the warehouse as a graph. We do not restrict the modeling to fact and dimension tables along with their interrelationships and any available views, but we extend the modeling to incorporate all the elements of an information system. To this end, *we add queries as integral parts of the configuration of a data warehouse*. In practice, a typical database is surrounded by forms, reports, web pages, stored procedures, and triggers deployed on the database server. Each of these software artifacts hides a list of queries via which it communicates with the database and exchanges queries and data with it. In addition, a data warehouse comprises a set of extract-transform-load (ETL) scripts, necessary for its population and refreshment with fresh source data. Queries constitute a convenient abstraction that captures the "skeleton" of all these applications w.r.t. their interrelationship to the database. We model the whole environment as a graph, with relations, attributes, constraints, queries, and query operands being the nodes of the graph, while the part-of or querying relationships are modeled as the edges connecting these nodes.

Second, our treatment for the evolution of the warehouse over time is based on events such as 'rename measure', 'add dimension attribute', 'delete dimension table', and so on. All these events are applied over the corresponding node and propagated over the appropriate subset of the graph. This way, given an evolution event, we can detect all the affected nodes. Moreover, we can define policies to regulate how a node will react to the possible change; e.g., a node can block -veto- an event, state the deletion of a dimension table, and isolate subsequent software modules that depend upon it from the effects of the change. We have built a what-if analysis tool that assesses potential evolution scenarios based on the above principles.

Based on these two pillars (detailed in Sections 2 and 3, respectively), in this paper, we provide a set of metrics for the assessment of the vulnerability of all the design structures in a data warehouse environment (Section 4). We exploit the graph and provide metrics like the *degrees* (in, out, and total) *of a node*, the *transitive degrees of a node* (standing for the extent to which other nodes transitively depend upon it), and the *degrees of a summarized variant of a module* (e.g., a view) that abstract the internal semantics of the module and focus on its coupling to the rest of the environment. We also provide an information theoretic definition of a *module's entropy* that simulates the extent to which the vulnerability of a node is surprising. Finally, we extensively experiment with various

configurations in the setup of a reference warehouse (Section 5) and assess both the effectiveness of the proposed metrics (i.e., how well do they actually predict the impact of evolution events to a design construct) and how different design alternatives for the same schema behave w.r.t. evolution.

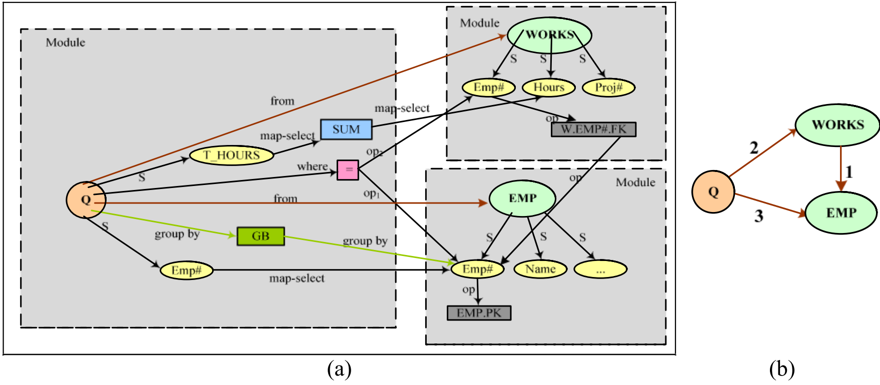
2 Graph Based Modeling for Data Warehouses

In this section, we summarize our graph modeling technique that uniformly covers relational tables, views, ETL activities, database constraints, and SQL queries as first class citizens. The proposed modeling technique represents all the aforementioned database parts as a directed graph *G*=(*V*,*E*). The nodes represent the entities of our model and the edges represent the relationships among these entities. Originally, the model was introduced in [12] and here, we provide only a short summary.

Each **relation** $R(\Omega_1,\Omega_2,...,\Omega_n)$ in the database schema is represented as a directed graph, which comprises: (a) a *relation node*, R, representing the relation 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.

The graph representation of a Select - Project - Join - Group By (SPJG) **query** involves a new node representing the query, named *query node*, and *attribute nodes* corresponding to the schema of the query. The query graph is a directed graph connecting the query node with all its schema 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. The edges connected the involved attribute and operand nodes are annotated as *map-select*, *from*, and *where relationships.* Aliases in the FROM clause (mostly needed in self-joins for our modeling) are annotated with *alias* edges. The direction of the edges is from the query node to the attribute nodes. WHERE and HAVING clauses are modeled via a left-deep tree of logical operands to represent the selection formulae; all the involved edges are annotated as *where* and *having relationships*, respectively. Nested queries are part of this modeling, too.

For the representation of aggregate queries, we employ two special purpose nodes: (a) a new node denoted as $GB \in \mathbf{GB}$, 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 query node towards the GB node that are labeled <group-by>, indicating *group-by relationships*, $\mathbf{E_G}$. Then, the GB node is connected with each of the aggregators through an edge tagged also as <group-by>, directing from the GB node towards the respective attributes. These edges 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 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 relation attribute.



(a) (b)

Fig. 1. (a) Graph and (b) abstract representation of an example aggregate query [12]

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 used in queries are denoted as a special purpose node $F_i \in \mathbf{F}$ having 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). **Views** are considered either as queries or relations (materialized views). An **ETL activity** is modeled as a sequence of SQL views. *DML statements* are denoted as queries. Fig. 1 depicts the proposed graph representation for the following query:

```
Q: SELECT EMP.Emp# as Emp#, Sum(WORKS.Hours) as T_Hours
   FROM EMP,WORKS WHERE EMP.Emp#=WORKS.Emp# GROUP BY EMP.Emp#
```

Modules. A module is a sub-graph of the overall graph in one of the following patterns: (a) a relation with its attributes and all its constraints, (b) a view with its attributes, functions and operands, and (c) a query with all its attributes, functions and operands. Modules are disjoint with each other and connected through edges concerning foreign keys, map-select and so on. Within a module, we distinguish *top-level* and *low-level* nodes. Top level nodes are used to signify the identity of the module; for that purpose, query, relation and view nodes are used as top-level nodes. Low-level nodes comprise the rest of the module. Edges are classified into *provider* and *part-of* relationships. Provider edges are intermodule relationships (e.g., $\mathbf{E_M}$, $\mathbf{E_F}$), whereas part-of edges are intramodule relationships (e.g., $\mathbf{E_S}$, $\mathbf{E_W}$). In Fig. 1, the graph comprises 3 modules corresponding to the query and the relations subgraphs.

Zoom in/out. Abstracting the graph into a modular representation at a coarser level of detail (zoom-out) involves the following steps: (a) for each query, view or relation module, all low-level nodes and intramodule edges are suppressed and only the respective top-level node is retained, and, (b) all inter-module edges apart from *from* and *foreign key* edges are dropped. A surviving edge between two modules is annotated with a weight corresponding to the number of the edges that originally connected the two modules. We call this weight the *strength* of the edge as it assesses

how tightly the involved modules are coupled. Fig. 1(b) depicts the abstract modular representation of Fig. 1(a).

3 Evolution in Data Warehouses

Data warehouse evolution is about changes and means to handle occurring changes.

Events. In our setting, we assume the following classes of occurring events:

C1. A dimension is removed, or renamed (`DEL`, `UPD Dimension Table`)
C2. The structure of a dimension table is updated (`ADD`, `DEL`, `UPD Dimension Attribute`)
C3. A fact table is completely decoupled from a dimension (`DEL FK`), or decoupled from one dimension and coupled to another (`UPD FK`)
C4. The measures of a fact table change (`ADD`, `DEL`, `UPD measure`)

An update can signify a change of data types or a renaming of a construct (our practical experience [12] indicates that it mostly refers to the latter.) We do not check for additions of fact or dimension tables, because such events do not result in a direct impact to any other logical warehouse construct per se. Given these changes that can occur to a data warehouse, their basic impact is that all software modules that use these database structures must be rewritten. The impact can be both syntactic (in the sense that all views and queries using a deleted attribute will crash) and semantic (in the sense that a new attribute in a relation or a modified condition in a view might require a rewriting of all the queries that use it). Assume for example that an attribute *FullName* is split to attributes *FirstName* and *LastName* or a view condition '*Year = 2007*' is altered to '*Year* > 2006'. The former change has syntactic impacts to all the queries using the attribute and the latter has semantic impact, since some of the queries using the view require exactly values of 2007, whereas some others will serve the purpose with any value greater than 2006.

Handling of events. Given an event posed to one of the warehouse constructs (or, equivalently, to one of the nodes of the graph of the warehouse that we have introduced), the impact involves the possible rewriting of the constructs that depend upon the affected construct either directly, or transitively. In a non-automated way, the administrator has to check all of these constructs and restructure the ones he finds appropriate. This process can be semi-automated by using our graph-based modelling and annotating the nodes and the edges of the graph appropriately with policies in the event of change. Assume for example, that the administrator guarantees to an application developer that a view with the sum of sales for the last year will always be given. Even if the structure of the view changes, the queries over this view should remain unaffected to the extent that its `SELECT` clause does not change. On the contrary, if a query depends upon a view with semantics '*Year = 2007*' and the view is altered to '*Year* > 2006', then the query must be rewritten.

The main idea in our approach involves annotating the graph constructs (relations, attributes, and conditions) sustaining evolution changes (addition, deletion, and modification) with policies that dictate the way they will regulate the change. Three kinds of policies are defined: (a) *propagate* the change, meaning that the graph must be

reshaped to adjust to the new semantics incurred by the event; (b) *block* the change, meaning that we want to retain the old semantics of the graph and the hypothetical event must be vetoed or, at least, constrained, through some rewriting that preserves the old semantics; and (c) *prompt* the administrator to interactively decide what will eventually happen. In [13] we have proposed a language that greatly alleviates the designer from annotating each node separately and allows the specification of default behaviors at different levels of granularity with overriding priorities. Assume that a default behavior for the deletion of view attributes is specified via the language of [13]. This policy can later be overridden with a directive for the behavior of the attributes of view *V* (again via the same language). Again, this policy can in turn be overridden with a specification for the behavior of attribute *V.A*.

Given the annotation of the graph, there is also a simple mechanism that (a) determines the status of a potentially affected node on the basis of its policy, (b) depending on the node's status, the node's neighbors are appropriately notified for the event. Thus, the event is propagated throughout the entire graph and affected nodes are notified appropriately. The `STATUS` values characterize whether (a) a node or one of its children (for the case of top-level nodes) is going to be deleted or added (e.g., `TO-BE-DELETED`, `CHILD-TO-BE-ADDED`) or (b) the semantics of a view have changed, or (c) whether a node blocks the further propagation of the event (e.g., `ADDITION-BLOCKED`).

4 Metric Suite

Various approaches exist in the area of database metrics. Most of them attempt to define a complete set of database metrics and map them to abstract quality factors, such as maintainability, good database design, and so on. In this section, we introduce a metric set based on the properties of the warehouse graph for measuring and evaluating the design quality of a data warehouse with respect to its ability to sustain changes. Metrics are based on properties of the aforementioned graph model.

4.1 Degree-Related Metrics

The first family of metrics comprises simple properties of each node in the graph. The main idea lies in the understanding that the in-degree, out-degree and total degree of a node *v* demonstrate in absolute numbers the extent to which (a) other nodes depend upon *v*, (b) the dependence of *v* to other nodes and (c) *v* is interacting with other nodes in the graph, respectively. Specifically, these metrics are:

- *In-degree*, $D^I(v)$, *Out-degree*, $D^O(v)$, *Degree,* $D(v)$, of a node *v*, with the simple semantics that have already been mentioned. These metrics have been introduced in [15] and assess the dependence and the responsibility of each node.
- *In Transitive*, *Out Transitive*, *Transitive degree.* The simple degree metrics of a node *v* are good measures for finding the nodes that are directly dependent on *v* or on which *v* directly depends on, but they cannot detect the transitive dependencies between nodes. Thus, if we consider the graph $G(V,E)$, the transitive degrees of a node $v \in V$ with respect to all nodes $y_i \in V$ are given by the following formulae:

$TD^{I}(v) = \sum_{y_i \in V} \sum_{p \in paths(y_i,v)} count(e_p)$, for all distinct edges $e_p \in paths$ of the form (y_i,v)

$TD^{O}(v) = \sum_{y_i \in V} \sum_{p \in paths(v,y_i)} count(e_p)$, for all distinct edges $e_p \in paths$ of the form (v,y_i)

$$TD(v) = TD^{I}(v) + TD^{O}(v)$$

– *Zoomed-out degree*. Assuming the degrees of the detailed graph can be computed, one can measure the degrees of the nodes of the zoomed-out graph. As already mentioned in section 2, zooming-out annotates edges with strengths, so the following formulae can be defined:

$D^{Is}(v) = \sum_{i} strength(e_i)$, for all edges e_i of the form (y,v)

$D^{Os}(v) = \sum_{i} strength(e_i)$, for all edges e_i of the form (v,y)

$$D^{s}(v) = D^{Is}(v) + D^{Os}(v)$$

– *Zoomed-out transitive degree*: Similarly to above, we may extend the transitive degrees to the zoomed-out graph, so the following formulae can be defined:

$TD^{Is}(v) = \sum_{y_i \in V} \sum_{p \in paths(y_i,v)} strength(e_p)$, for $e_p \in paths$ of the form (y_i,v)

$TD^{Os}(v) = \sum_{y_i \in V} \sum_{p \in paths(v,y_i)} strength(e_p)$, for $e_p \in paths$ of the form (v,y_i)

$$TD^{s}(v) = TD^{Is}(v) + TD^{Os}(v)$$

There are several other variants of these graph-based measures that we do not explore here. We can define *Category-constrained degrees*, which constrain degrees by edge categories. For example, we might be interested only in the number of part-of outgoing edges of a relation. We can also measure the importance of modules (e.g., using the frequency of a query's execution) and obtain *weighted variants* of the aforementioned metrics.

4.2 Entropy – Based Metrics

Entropy is used to evaluate the extent to which a part of a system is less likely to be affected by an evolution event than other parts [1]. Given a set of events $A=[A_1,..., A_n]$ with probability distribution $P=\{p_1,…,p_q\}$, respectively, entropy is defined as the average information obtained from a single sample from A:

$$H(A) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Assume a node v in our graph $G(V,E)$. We define the probability that $v \in V$ is affected by an arbitrary evolution event e over a node $y_k \in V$ as the number of paths from v towards y_k divided by the total paths from v towards all nodes in the graph, i.e.,

$$P(v|y_k) = \frac{paths(v, y_k)}{\sum_{y_i \in V} paths(v, y_i)}, \text{ for all nodes } y_i \in V.$$

The information we gain when a node v is affected by an event occurred on node y_k is $I(P(v \mid y_k)) = \log_2 \frac{1}{P(v \mid y_k)}$ and the entropy of node v wrt the whole graph is then:

$$H(v) = -\sum_{y_i \in V} P(v \mid y_i) \log_2 P(v \mid y_i), \text{ for all nodes } y_i \in V.$$

The above quantity expresses the average information we gain, or equivalently the amount of “surprise” conveyed, if node v is affected by an arbitrary evolution event on the graph. Observe that high entropy values correspond to nodes with a higher dependence with the rest of the graph. For instance, a query defined over only one relation has an entropy value of 0, whereas a query defined over a view which in turns accesses two relations has an entropy value of $\log_2 3$.

Moreover, we can apply the exact same technique to the zoomed out-graph $G^s(V^s,E^s)$, by defining the probability of a node $v \in V^s$ to be affected by an evolution event over a node $y_k \in V^s$ as:

$$P^s(v|y_k) = \frac{\sum_{p \in paths(v,y_k)} strength(e_p)}{\sum_{y_i \in V^s} \sum_{p \in paths(v,y_i)} strength(e_p)}, \text{ for all nodes } y_i \in V^s.$$

with $e_p \in E^s$ being the edges of all the paths of the zoomed out graph stemming from v towards y_k. Similarly, the entropy of node $v \in V^s$ is:

$$H^s(v) = -\sum_{y_i \in V^s} P^s(v \mid y_i) \log_2 P^s(v \mid y_i), \text{ for all nodes } y_i \in V^s.$$

5 Evaluation – Experiments

Goals. There are two major goals in our experiments. First, we have investigated the extent to which the proposed metrics good indicators for the prediction of the effect evolution events have on the warehouse. A clear desideratum in this context is the determination of the most suitable metric for this prediction under different circumstances. A second goal involves the comparison of alternative design techniques with respect to their tolerance to evolution events.

Experimental setup for the first goal. To achieve the goal of determining the fittest prediction metric, we need to fix the following parameters: (a) a data warehouse schema surrounded by a set of queries and possibly views, (b) a set of events that alter the above configuration, (c) a set of administrator profiles that simulate the intention of the administrating team for the management of evolution events, and (d) a baseline method that will stand as an accurate estimate of the actual effort needed to maintain the warehouse environment.

We have employed the TPC-DS [14] schema as the testbed for our experiments. TPC-DS is a benchmark that involves six star schemas (with a large overlap of shared dimensions) standing for Sales and Returns of items purchased via a Store, a Catalog and the Web. We have used the Web Sales schema that comprises one fact table and thirteen dimension tables. The structure of the Web Sales schema is interesting in the sense that it is neither a pure star, nor a pure snowflake schema. In fact, the dimensions are denormalized, with a different table for each level; nevertheless, the fact table has foreign keys to all the dimension tables of interest (resulting in fast joins with the appropriate dimension level whenever necessary). Apart from this "starified" schema, we have also employed two other variants in our experiments: the first involves a set of views defined on top of the TPC-DS schema and the second involves the merging of all the different tables of the Customer dimension into one. We have isolated the queries that involve only this subschema of TPC-DS as the surrounding query set of the warehouse. The views for the second variant of the schema were determined by picking the most popular atomic formulae at the WHERE clause of the surrounding queries. In other words, the aim was to provide the best possible reuse of common expressions in the queries.

We created two workloads of events to test different contexts for the warehouse evolution. The first workload of 52 events simulates the percentage of events observed in a real world case study in an agency of the Greek public sector. The second workload simulates a sequence of 68 events that are necessary for the migration of the current TPC-DS Web sales schema to a pure star schema. The main idea with both workloads is to simulate a set of events over a reasonable amount of time. Neither the internal sequence of events per se, nor the exact background for deriving the events is important; but rather, the focus is on the events' generation that statistically capture a context under which administration and development is performed (i.e., maintenance of the same schema in the first case, and significant restructuring of a schema in the latter case). The distribution of events is shown in Table 1.

We have used an experimental prototype, HECATAEUS [11], for the identification of the impact of hypothetical evolution events. We have annotated the graph with policies, in order to allow the management of evolution events. We have used three annotation "profiles", specifically: (a) *propagate all*, meaning that every change will be flooded to all the nodes that should be notified about it, (b) *block all*, meaning that a view/query is inherently set to deny any possible changes, and (c) *mixture*, consisting of 80% of the nodes with propagate policies and 20% with blocking. The first policy practically refers to a situation without any annotation. The second policy simulates a highly regulatory administration team that uses HECATAEUS to capture an evolution event as soon as it leaves its source of origin; the tool highlights the node where the event was blocked. The third policy simulates a rather liberal environment, where most events are allowed to spread over the graph, so that their full impact can be observed; yet, 20% of critical nodes are equipped with blocking policies to simulate the case of nodes that should be handled with special care.

Summarizing, the configuration of an experiment involves fixing a schema, a set of policies and a workload. We have experimented with all possible combinations of values. The measured metrics in each experiment involve the execution of the workload of evolution events in the specified configurations and the measurement of the

Table 1. Distribution of events

Operation	Distribution 1	Distribution 2
Rename Measure	29% (15)	0% (0)
Add Measure	25% (13)	0% (0)
Rename Dimension Attribute	21% (11)	0% (0)
Add Dimension Attribute	15% (8)	37% (25)
Delete Measure	6% (3)	0% (0)
Delete Dimension Attribute	4% (2)	44% (30)
Delete FKs	0%	13% (9)
Delete Dimension Table	0%	6% (4)

affected nodes. Specifically, each node of the graph is monitored and we get analytic results on how many times each node was affected by an event. This measurement constitutes the baseline measurement that simulates what would actually happen in practice. This baseline measurement is compared to all the metrics reported in Section 4, being evolution-agnostic or not.

Experimental Setup for the second goal. The second goal of our experiments is to compare alternative designs of the warehouse with each other – i.e., we want to find which design method (pure star, TPC-DS with or without views) is the best for a given designer profile (which is expressed by the policies for the management of evolution). Thus, the comparison involves the compilation of the baseline measurements, grouped per policy profile and alternative schema. We measure the total number of times each node was affected and we sum all these events. The intention is to come up with a rough estimation of the number of rewritings that need to be done by the administrators and the application developers (in this setting, it is possible that a query or view is modified in more than one of its clauses). A second measurement involves only the query part: we are particularly interested in the effort required by the application developers (which are affected by the decisions of the administration team), so we narrow our focus to the effect inflicted to the queries only.

5.1 Effectiveness of the Proposed Metrics

In this experiment, we evaluate the effectiveness of the proposed metrics using the first distribution of events. We have constructed the following nine configurations by fixing each time a value for the schema and the policy. The schema takes one of the values {Web Sales (*WS*), Web Sales extended with views (*WS-views*), star variant of Web Sales (*WS-star*)} and the policy takes one of the values {*Block-All*, *Propagate-All*, *Mixture*}. In the rest, we discuss our findings organized in the following categories: (a) Fact Tables, (b) Dimension Tables, (c) Views, and (d) Queries.

Facts. Our experiments involved a single fact table. We observed that the number of events that occurred to the fact table does not change with the overall architecture. The presence of more or less dimensions or views did not affect the behavior of the fact table; on the contrary, it appears that the main reasons for the events that end up to the fact table, are its attributes. Therefore, the main predictor for the behavior of the evolution of the fact table is its out-degree, which is mostly due to the part-of relationships with its attributes.

Dimension Tables. Evolution on dimension tables can also be predicted by observing their out-degree, since this property practically involves the relationship of the dimension with its attributes as well as its relationship via foreign keys with other dimensions. Figure 2 depicts this case for the original web sales schema and its star variant, for which all customer-related dimensions have been merged into one dimension. Our baseline (depicted as a solid line with triangles) involves the actual number of times a node belonging to a dimension table was affected.

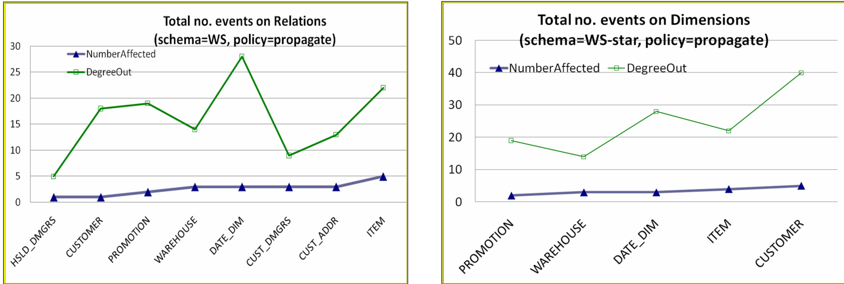


Fig. 2. Events affecting dimensions: (a) WS schema, (b) WS-star schema

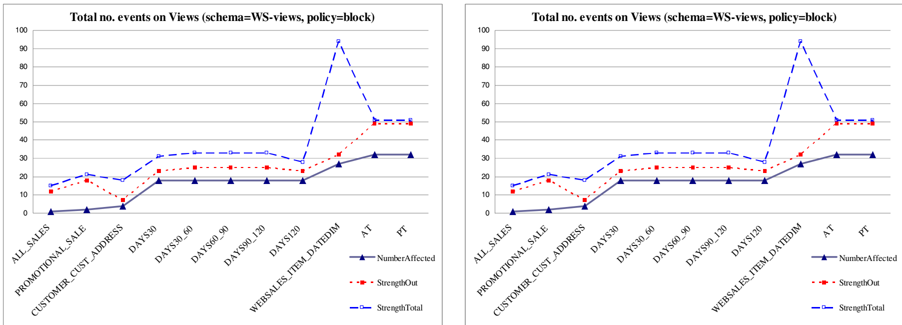


Fig. 3. Events affecting views: (a) WS-star and WS schema, (b) WS-views schema

Despite the spikes at the heavily correlated date dimension, out-degree is a predictor, keeping in mind that it is the actual trend that matters and not the values themselves.

Views. Views behave practically uniformly for all configurations, independently of schema or policy. Observe Fig. 3 where we depict our findings concerning views. It is clear that strength of out-degree (strength-out) and total strength are the best predictors for the evolution of views with the former being an interestingly accurate predictor in all occasions. Figure 3(a) is a representative of all the six configurations for the original web sales schema and its star variant. The policy makes no difference and all six experiments have resulted in exactly the same behavior. The rest of the metrics miss the overall trend and are depicted for completeness. Fig. 3(b) shows a representative graphical representation of the metrics, showing that the strength of the out-degree is consistently effective, whereas the total strength shows some spikes (mainly due to views that are highly

connected to the sources, although these sources did not generate too much traffic of evolution events after all). The rest of the metrics behave similarly with Fig. 3(a).

Queries. Queries are typically dependent upon their coupling to the underlying DBMS layer. As a general guideline, the most characteristic measure of the vulnerability of queries to evolution events is their transitive dependence. A second possible metric suitable for a prediction is the entropy; however, it is not too accurate. Other metrics do not seem to offer good prediction qualities; the best of them, out-degree, does not exceed 70%. Recall that the baseline for our experiment is the actual number of events that reached a query (depicted as a solid line decorated with triangles in Fig. 4 and 5). Finally, we stress that the trend makes a metric successful and not the precise values.

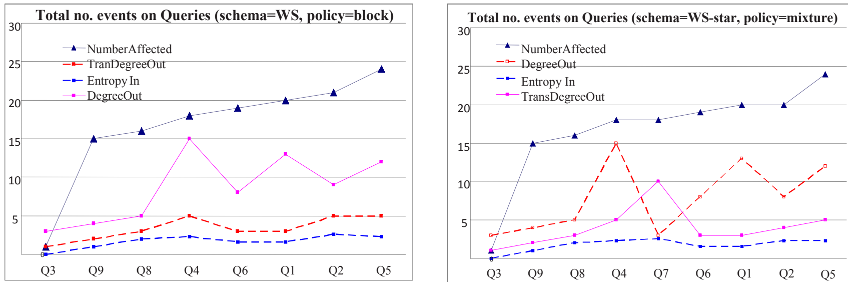


Fig. 4. Events affecting queries: (a) WS schema, (b) WS-star schema

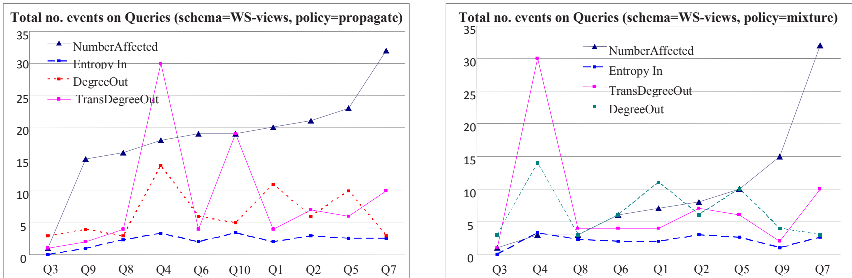


Fig. 5. Total number of events affecting queries: (a) Behavior for the WS-views with propagate policy; (b) Behavior for the WS-views schema with mixture policy

Fig. 4 shows two characteristic plots for the original web sales schema and its star variant. Each plot is a representative of the other plots concerning the same schema, with the trends following quite similar behavior. In all cases, transitive dependence gives a quite successful prediction, with around 80% accuracy. It is noteworthy that in the case of the 20% of failures, though, the metric identifies a query as highly vulnerable and in practice, the query escapes with few events. Fortunately, the opposite does not happen, so a query is never underestimated with respect to its vulnerability. Entropy is the second best metric and due to its smoothness, although it follows transitive dependence's behavior, it misses the large errors of transitive dependence, although it also misses the scaling of events, for the same reason.

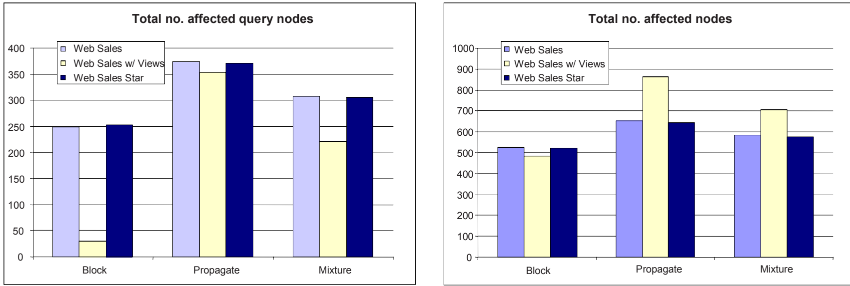


Fig. 6. Comparison of *WS, WS-views*, *WS-star* design configurations for distribution 1: (a) only affected queries and (b) all affected nodes

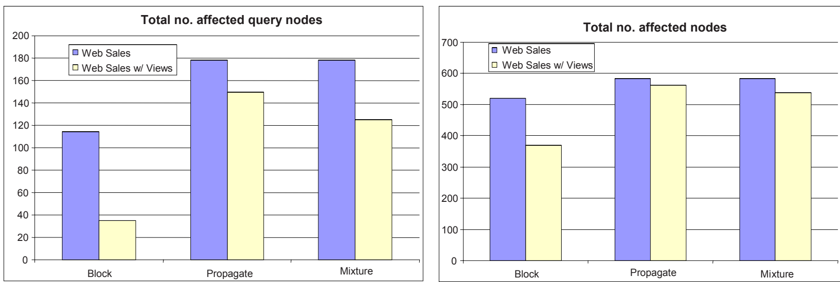


Fig. 7. Comparison of *WS, WS-views* design configurations for distribution 2: (a) only affected queries; (b) all affected nodes

Queries are quite dependent on the policy and schema: views seem to block the propagation of events to the queries. Fig. 5(b) shows a significant drop for the values of affected queries when the policy is a mixture of propagation and blocking policies. The propagate-all policy depicted in Fig. 5(a) presents the flooding of the events, which involves more than double the number of occurrences as compared to the numbers of Fig. 5(b) for 80% of the cases. A block-all policy involved only 3 of the 10 queries and it is not depicted for lack of space). Interestingly, the transitive degree has a success ratio of 80%, as opposed to the rather unsuccessful out-degree.

5.2 Comparison of Alternative Design Configurations

We compared the three alternative design configurations of our system in order to come up with an estimation of the number of rewritings that need to be done by the administrators and the application developers, and to assess the effect that a different schema configuration has on the system. Thus, we measured the number of affected nodes and specifically, the number of affected query nodes for the nine different configurations of policy sets and schemata. The first distribution of events was applied to all schemas, whereas the second was applied only to *WS* and *WS-views*.

Fig. 6 describes the effect that a design alternative has on how affected system constructs are in the case of evolution. A star schema has less maintenance effort than the other variants due to its reduced size. Clearly, the presence of views augments the

effort needed by the administration team to maintain them (shown in the increased number of affected nodes of Fig. 6b), which is because nodes belonging to views are extensively affected. Still, the interference of views between the warehouse and the queries serves as a "shield" for absorbing schema changes and not propagating them to queries. The drop in query maintenance due to the presence of views is impressive: *whatever we pay in administration effort, we gain in development effort*, since the cost of rewritings in terms of human effort mainly burdens application developers, who are obliged to adapt affected queries to occurred schema changes. The case of schema migration strengthens this observation (Fig. 7). As for the different policy sets, we observe that blocking of events decreases the number of affected nodes in all configurations and saves significant human effort. It is, however, too conservative, constraining even the necessary readjustments that must be actually made on queries and views. On the other hand, propagate and mixture policy sets have an additional overhead, which is balanced by the automatic readjustments that are held on the system.

6 Conclusions

In this paper, we have proposed a set of metrics for the evaluation of the vulnerability warehouse modules to future changes and for the assessment of the quality of alternative designs of the warehouse. We have learned that out-degrees help as predictors for the fact and the dimension tables of the warehouse; the strength of out-degree (strength-out) and total strength are very good predictors for the evolution of views; the transitive dependence and entropy are good predictors for the vulnerability of queries. As far as warehouse design is concerned, we have an elegant theory to characterize the trade-offs between administration and development costs that result from the choice of adding views or "starifying" the schema of a warehouse.

Further experimentation and novel metrics along with theoretical validation of the proposed ones are clear topics for future work.

References

1. Allen, E.B.: Measuring Graph Abstractions of Software: An Information-Theory Approach. In: METRICS (2002)
2. Bellahsene, Z.: Schema evolution in data warehouses. Knowl. and Inf. Syst. 4(2) (2002)
3. Berenguer, G., et al.: A Set of Quality Indicators and their Corresponding Metrics for Conceptual Models of Data Warehouses. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2005. LNCS, vol. 3589. Springer, Heidelberg (2005)
4. Blaschka, M., Sapia, C., Höfling, G.: On Schema Evolution in Multidimensional Databases. In: Mohania, M., Tjoa, A.M. (eds.) DaWaK 1999. LNCS, vol. 1676. Springer, Heidelberg (1999)
5. Fan, H., Poulovassilis, A.: Schema Evolution in Data Warehousing Environments - A Schema Transformation-Based Approach. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288. Springer, Heidelberg (2004)
6. Favre, C., Bentayeb, F., Boussaid, O.: Evolution of Data Warehouses' Optimization: A Workload Perspective. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2007. LNCS, vol. 4654. Springer, Heidelberg (2007)

7. Golfarelli, M., Lechtenbörger, J., Rizzi, S., Vossen, G.: Schema versioning in data warehouses: Enabling cross-version querying via schema augmentation. Data Knowl. Eng. 59(2), 435–459 (2006)
8. Gupta, A., Mumick, I.S., Rao, J., Ross, K.: Adapting materialized views after redefinitions: Techniques and a performance study. Information Systems (26) (2001)
9. Levene, M., Loizou, G.: Why is the snowflake schema a good data warehouse design? Information Systems Journal 28(3), 225–240 (2003)
10. Nica, A., Lee, A.J., Rundensteiner, E.A.: The CSV 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. Springer, Heidelberg (1998)
11. Papastefanatos, G., Anagnostou, F., Vassiliadis, P., Vassiliou, Y.: Hecataeus: A What-If Analysis Tool for Database Schema Evolution. In: CSMR (2008)
12. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: What-if Analysis for Data Warehouse Evolution. In: Song, I.-Y., Eder, J., Nguyen, T.M. (eds.) DaWaK 2007. LNCS, vol. 4654. Springer, Heidelberg (2007)
13. Papastefanatos, G., et al.: Language Extensions for the Automation of Database Schema Evolution. In: ICEIS (2008)
14. The TPC BENCHMARK™ DS (April 2007), `http://www.tpc.org/tpcds/spec/tpcds1.0.0.d.pdf`
15. Vassiliadis, P., Simitsis, A., Skiadopoulos, S.: Modeling ETL activities as graphs. In: DMDW (2002)
16. Wedemeijer, L.: Defining Metrics for Conceptual Schema Evolution. In: FMLDO (2000)