# Schema Evolution for Databases and Data Warehouses

Petros Manousis[1], Panos Vassiliadis[1(✉)], Apostolos Zarras[1], and George Papastefanatos[2]

[1] Department of Computer Science, University of Ioannina (Ioannina, Hellas), Ioannina, Greece
{pmanousi,pvassil,zarras}@cs.uoi.gr
[2] Athena Research Center (Athens, Hellas), Athens, Greece
gpapas@imis.athena-innovation.gr

**Abstract.** Like all software systems, databases are subject to evolution as time passes. The impact of this evolution is tremendous as every change to the schema of a database affects the syntactic correctness and the semantic validity of all the surrounding applications and de facto necessitates their maintenance in order to remove errors from their source code. This survey provides a walk-through on different approaches to the problem of handling database and data warehouse schema evolution. The areas covered include (a) published case studies with statistical information on database evolution, (b) techniques for managing schema and view evolution, (c) techniques pertaining to the area of data warehouses, and, (d) prospects for future research.

## 1 Introduction

Evolution of software and data is a fundamental aspect of their lifecycle. In the case of data management, evolution concerns changes in the contents of a database and, most importantly, in its schema. Database evolution can concern (a) changes in the operational environment of the database, (b) changes in the content of the databases as time passes by, and (c) changes in the internal structure, or *schema*, of the database. *Schema evolution*, itself, can be addressed at (a) the conceptual level, where the understanding of the problem domain and its representation via an ER schema evolves, (b) the logical level, where the main constructs of the database structure evolve (for example, relations and views in the relational area, classes in the object-oriented database area, or (XML) elements in the XML/semi-structured area), and, (c) the physical level, involving data placement and partitioning, indexing, compression, archiving etc.

In this survey, we will focus on the evolution of the logical schema of relational data and also extend our survey to the special case of data warehouse evolution. For the rest, we refer the interested reader to the following very interesting surveys. First, it is worth visiting a survey by Roddick [1], which appeared 20 years ago and summarizes the state of the art of the time in the areas of schema versioning and evolution, with emphasis to the modeling, architectural and query

© Springer International Publishing Switzerland 2016
E. Zimányi and A. Abelló (Eds.): eBISS 2015, LNBIP 253, pp. 1–31, 2016.
DOI: 10.1007/978-3-319-39243-1_1

language issues related to the support of evolving schemata in database systems. Second, 16 years later, a comprehensive survey by Hartung, Terwilliger and Rahm [2] appeared, in which the authors classify the related tools and research efforts in the following subareas: (a) the management of the evolution of relational database schemata, (b) the evolution of collections of XML documents, and (c) the evolution of ontologies. In the web site http://dbs.uni-leipzig.de/en/publications one may also find a comprehensive list of publications in the broader area of schema and data evolution. From our part, the material that we survey is collected by exploiting three sources of information: (a) our own monitoring of the field over the years, (b) by building on top of the aforementioned surveys, and, (c) by inspecting the main database and business intelligence venues in the last years, to identify the new works that have taken place since the last survey.

We organize the presentation of the material as follows. In Sect. 2, we discuss empirical studies in the area of database evolution. In Sect. 3, we present the sate of practice. In Sect. 4, we cover issues related to the identification of the impact that database evolution has to external applications and queries, as well as to views. In Sect. 5, we cover the specific area of data warehouses from the viewpoint of evolution. We conclude with thoughts around open issues in the research agenda in the area of evolution in Sect. 6.

## 2 Empirical Studies on Database Evolution

In this section, we survey empirical studies in the area of database evolution. These studies monitor the history of changes and report on statistical properties and recurring phenomena. In our coverage we will follow a chronological order, which also allows us to put the studies in the context of their time.

### 2.1 Statistical Profiling of Database Evolution via Real-World Studies

**Studies During the 1990's.** The first account of a sizable empirical study, by Sjoberg [3], discusses the evolution of the database schema of a health management system over a period of 18 months, monitored by a tool specifically constructed for this purpose. A single database schema was examined, and, interestingly, the monitored system was accompanied by a metadata dictionary that allowed to trace how the queries of the applications surrounding the database relate to the tables and attributes of the evolving database. Specific numbers for the evolution of the system, during this period of 18 months, include:

- There was a 139% increase of the number of tables and a 274% increase of the number of attributes (including affected attributes due to table evolution), too.
- All (100%) the tables were affected by the evolution process.
- Additions were more than deletions, by an 28% tables and a 42% for attributes.

- An insignificant percentage of alterations involved renaming of relations or merge/split of tables.
- Changes in the type of fields (i.e., data type, not null, unique constraints) proved to be equal to additions (31 both) and somehow less than deletions (48) for a period of 12 months, during which this kind of changes were studied.
- On average, each relation addition resulted in 19 changes in the code of the application software. At the same time, a relation deletion produced 59.5 changes in the application code. The respective numbers for attributes were 2 changes for attribute additions and 3.25 changes for attribute deletions, respectively.
- The evolution process was characterized by an inflating period (during construction) where almost all changes were additions, and a subsequent period where additions and deletions where balanced.

**Revival in Late 2000's.** In terms of empirical studies, and to the best of our knowledge, no developments took place for the next 15 years. This can be easily attributed to the fact that the research community would find it very hard to obtain access to monitor database schemata for an in-depth and long study. The proliferation of free and open-source software changed this situation. So, in the last few years, there are more empirical studies in the area that report on how schemata of databases related to open source software have evolved.

The first of these studies came fifteen years later after the study of Sjoberg. The authors of [4] made an analysis on the database back-end of MediaWiki, the software that powers Wikipedia. The study conducted over the versions of four years, and came with several important findings. The study reports an increase of 100 % in the number of tables and a 142 % in the number of attributes. Furthermore, 41.5 % of the attributes of the original database were removed from the database schema, and 25.1 % of the attributes were renamed respectively. The major reasons for these alterations were (a) the improvement of performance, which in many cases induces partitioning of existing tables, creation of materialized views, etc., (b) the addition of new features which induces the enrichment of the data model with new entities, and (c) the growing need for preservation of database content history. A very interesting observation is that around 45% of changes do not affect the information capacity of the schema, but they are rather index adjustments, documentation, etc. A statistical study of change breakdown revealed that attribute addition is the most common alteration, with 39 % of changes, attribute deletion follows with 26 %, attribute rename was up to the 16 % and table creation involved a 9 % of the entire set of recorded changes. The rest of the percentages were insignificant.

Special mention should be made to this line of research [5], as the people involved in this line of research should be credited for providing a large collection of links[1] for open source projects that include database support. Also, it is worth mentioning here that the effort is related to PRISM (later re-engineered to PRISM++ [6]), a change management tool, that provides a language of Schema

[1] http://yellowstone.cs.ucla.edu/schema-evolution/index.php/Benchmark_Extension.

Modification Operations (SMO) (that model the creation, renaming and deletion of tables and attributes, and their merging and partitioning) to express schema changes (see Sect. 4.1 for details).

Shortly after, two studies from the Univ. of Riverside appear. In [7], Lin and Neamtiu study two aspects of database evolution and their effect to surrounding applications. The first part of the study concerns the impact that schema evolution has on the surrounding applications. The authors work with two cases, specifically the evolution of Mozilla, between 2005 and 2009 and the evolution of the Monotone version control system between 2003 and 2009, both of which use a database to store necessary information for their correct operation. The authors document and exemplify how the developers of the two systems address the issue of schema evolution between different versions of their products. The authors also discuss the impact of erroneous database evolution, even though there exists software that is responsible for the migration of the system's modules to the new database schema. One very interesting finding is that although the applications can include a check on whether the database schema is synchronized to the appropriate version of the application code, this check is not omnipresent; thus, there exist cases where the application can operate on a different schema than the one of the underlying database, resulting in crashes or data loss. At the same time, the authors have measured the breakdown of changes during the period that they have studied. The second part of the study concerns DBMS evolution (attention: *DBMS*, not database) from the viewpoint of file storage. The authors study SQLite, MySQL and Postgres on how different releases come with different file formats and how usable old formats can be under a new release of the DBMS. Also, the authors discuss how the migration of stored databases should be performed whenever the DBMS is upgraded, due to the non-compatibility of the file formats of the different releases.

In a similar vein, in [8], Wu and Neamtiu considered 4 case studies of embedded databases (i.e., databases tightly coupled with corresponding applications that rely on them) and studied the different kinds of changes that occurred in these cases. Specifically, the authors study the evolution of Firefox between 2004 and 2008, Monotone (a version management system) between 2003 and 2010, BiblioteQ (a catalog management suite) between 2008 and 2010 and Vienna (an RSS newsreader) between 2005 and 2010. Comparing their results to previous works, the authors see the same percentages concerning the expansion of the database, but a larger number of table and column deletions. This is attributed to the nature of the databases, as the databases that are studied by Wu and Neamtiu are embedded within applications, rather than largely used databases as in the case of the previous studies. Moreover, the authors performed a respective frequency and timing analysis, which showed that the database schemata tend to stabilize over time, as the evolution activity calms down over time. There is more change activity for the schemata at the beginning of their history, whereas the schemata seem to converge to a relatively fixed structure at later versions.

**A Large-Scale Study in 2013.** In [9], Qiu, Li and Su report on their study of the evolution of 10 databases, supporting open source projects. The authors collected the source files of the applications via their SVN repositories and isolated the changes to the logical schema of each database (i.e., they ignored changes involving comments, syntax correction, DBMS-related changes, and several others). The remaining changes are characterized by the authors as valid DB revisions. The authors report that they have avoided the automatic extraction of changes, as the automatic extraction misses changes like table split or merge, or renaming and have performed manual checks for all the valid DB revisions for all the datasets. The study covers 24 types of change including the additions and deletions of tables, attributes, views, keys, foreign keys, triggers, indexes, stored procedures, default value and not null constraints, as well as the renaming of tables, attributes and the change of data types and default values. We summarize the main findings of the study in four categories.

*Temporal and Locality Focus. Change is focused both (a) with respect to time and (b) with respect to the tables that change.* Concerning timing, a very important finding is that 7 out of 10 databases reached 60% of their schema size within 20% of their early lifetime. *Change is frequent in the early stages of the databases, with inflationary characteristics; then, the schema evolution process calms down.* Schema changes are also focused with respect to the tables that change: 40% of tables do not undergo any change at all, and 60%-90% of changes pertain to 20% of the tables (in other words, 80% of the tables live quiet lives). The most frequently modified tables attract 80% of the changes.

*Change breakdown.* The breakdown of changes revealed the following catholic patterns: (a) insertions are more than updates which are more than deletions and (b) table additions, column additions and data type changes are the most frequent types of change.

*Schema and Application Co-evolution.* To assess how applications and databases co-evolve, the authors have randomly sampled 10% of the valid database revisions and manually analyzed co-evolution. The most important findings of the study are as follows:

- First, the authors characterized the co-change of applications in four categories and assessed the breakdown of changes per category. In 16.22% of occasions, the code change was in a previous/subsequent version than the one where the database schema change occurred; 50.67% of application adaptation changes took place in the same revision with the database change, 21.62% of database changes were not followed by code adaptation and 11.49% of code changes were unrelated to the database evolution.
- A second result says that each atomic change at the schema level is estimated to result in 10 – 100 lines of application code been updated. At the same time, a valid database revision results in 100 – 1000 lines of application code being updated.

*A final note*: Early in the analysis of results, the authors claim that change is frequent in schema evolution of the studied datasets. Although we do not dispute the numbers of the study, we disagree with this interpretation: change

caries a lot between different cases (e.g., coppermine comes with 8.3 changes and 14.2 atomic changes per year contrasted to 65.5 changes and 299.3 atomic changes per year at Prestashop). We would argue that change can be arbitrary depending on the case; in fact, each database seems to present its own change profile.

### 2.2 Recent Advances in Uncovering Patterns in the Evolution of Databases

A recent line of research that includes [10–12], reveals patterns and regularities in the evolution of database schemata. At a glance, all these efforts analyze the evolution of the database schemata of 8 open source case studies. For each case study, the authors identified the changes that have been performed in subsequent schema versions and re-constructed the overall evolution history of the schema, based on Hecate, an automated change tracking tool developed by the authors for this purpose. The number of versions that have been considered for the different schemata ranged from 84 to 528, giving a quite rich data set for further analysis. Then, in [10] the authors perform a macroscopic study on the evolution of database schemata. Specifically, in this study the authors detect patterns and regularities that concern the way that the database schema grows over time, the complexity of the schema, the maintenance actions that take place and so on. To detect these patterns they resort to the properties that are described in Lehnman's laws of software evolution [13]. In [11], extend their baseline work in [10] with further results and findings revealed by the study, as long as detailed discussions concerning the relevance of the Lehman's laws in the case of databases, and the metrics that have been employed. On the other hand, in [12] the authors perform a microscopic study that delves into the details of the life of tables, including the tables' birth, death, and the updates that occur in between. This study reveals patterns, regularities and relations concerning the aforementioned aspects.

**The Life of a Database Schema.** In the early 70's, Lehman and his colleagues initiated their study on the evolution of software systems [14] and continued to refine and extend it for more than 40 years [13]. Lehman's laws introduce the properties that govern the evolution of *E-type systems*, i.e., software systems that solve a problem, or address an application in the real world [13]. For a detailed historical survey of the evolution of Lehman's laws the interested reader can refer to [15]. The essence of Lehman's laws is that *the evolution of an E-type system is a controlled process that follows the behavior of a feedback-based mechanism.* In particular, the evolution is driven by *positive feedback* that reflects the need to adapt to the changing environment, by *adding functionalities* to the evolving system. The growth of the system is constrained by *negative feedback* that reflects the need to perform *maintenance activities*, so as to prevent the deterioration of the system's quality.

In more detail, as discussed in [10,11] the laws can be organized in three groups that concern different aspects of the overall software evolution process.

The first group of laws discusses the existence of the feedback mechanism that constrains the uncontrolled evolution of software. The second group focuses on the properties of the growth part of the system, i.e., the part of the evolution mechanism that accounts for positive feedback. Finally, the third group of laws discusses the properties of perfective maintenance that constrains the uncontrolled growth, i.e., the part of the evolution mechanism that accounts for negative feedback. The major patterns and regularities revealed in [10,11] from the investigation of each group of laws are summarized below:

- *Feedback mechanism for schema evolution*: Overall, the authors found that schema evolution demonstrates the behavior of a stable, feedback-regulated system, as the need for expanding its information capacity to address user needs is controlled via perfective maintenance that retains quality; this antagonism restrains unordered expansion and brings stability. Positive feedback is manifested as expansion of the number of relations and attributes over time. At the same time, there is negative feedback too, manifested as housecleaning of the schema for redundant attributes or restructuring to enhance schema quality. In [10,11] the authors further observed that the inverse square models [16] for the prediction of size expansion hold for all the schemata that have been studied.
- *Growth of schema size due to positive feedback*: The size of the schema expands over time, albeit with versions of perfective maintenance due to the negative feedback. The expansion is mainly characterized by three patterns/phases, (i) abrupt change (positive and negative), (ii) smooth growth, and, (iii) calmness (meaning large periods of no change, or very small changes). The schema's growth mainly occurs with spikes oscillating between zero and non-zero values. Also, the changes are typically small, following a Zipfian distribution of occurrences, with high frequencies in deltas that involved small values of change, close to zero.
- *Schema maintenance due to negative feedback*: As stated in [11] the overall view of the authors is that due to the criticality of the database layer in the overall information system, maintenance is done with care. This is mainly reflected by the decrease of the schema size as well as the decrease in the activity rate and growth with age. Moreover, the authors observed that age results in a reduction of the complexity to the database schema. The interpretation of this observation is that perfective maintenance seems to do a really good job and complexity drops with age. Also, they authors point out that in the case of schema evolution, activity is typically less frequent with age.

**The Life of a Table - Microscopic Viewpoint.** In [12], the authors investigated in detail the relations between table schema size, duration and updates. The main findings of this study are summarized below:

- From a general perspective, early stages of the database life are more "active" in terms of births, deaths and updates, whereas, later, growth is still there, but deletions and updates become more concentrated and focused.

- The life and death of tables is governed by the *Gamma* pattern, which says that large-schema tables typically survive. Moreover, short-sized tables (with less than 10 attributes) are characterized by short durations. The deletions of these "narrow" tables typically take place early in the lifetime of the project either due to deletion or due to renaming (which is equivalent from the point of view of the applications: they crash in both cases).
- Concerning the amount of updates, most tables live quiet lives with few updates. The main reason is the dependency magnet phenomenon, i.e., table updates induce large impact on the surrounding dependent software.
- The relation between table duration and amount of updates is governed by the *inverse Gamma* pattern, which states that updates are not proportional to longevity, but rather, few top-changer tables attract most of the updates.
  - Top-changer tables live long, frequently they are created in the first version of the database and they can have large number of updates (both in absolute terms and as a normalized measure over their duration).
  - Interestingly top-changer tables, they are not necessarily the larger ones, but typically medium sized.

## 3 State of Practice

In this section, we discuss how the commercial database management systems handle schema changes. The systems that we survey are: (a) Oracle, (b) DB2 of IBM, and, (c) SQL Server and Visual Studio of Microsoft. Another part of this research is dedicated to the open sourced or academic tools that are dealing with the schema changes. Some of those tools are: (a) Django, (b) South, and, (c) Hecate.

### 3.1 Commercial Tools

**Oracle - Change Management Pack (CMP).** Oracle Change Management Pack ([17]) is part of Oracle Enterprise Manager. CMP enables the management and deployment of schema changes from development to production environments, as well as the identification of unplanned schema changes that potentially cause application errors.

CMP features the following concepts:

- Change plans: A change plan is an object that serves as a container for change requests.
- Baselines: A baseline is a group of database object definitions captured by the Create Baseline application at a particular point in time.
- Comparisons: A comparison identifies the differences found by the Oracle Change Management Pack in two sets of database object definitions that you have specified in the Compare Database Objects application.

The *Create Baseline* application enables users in creating database schema descriptions in a CMP format or plain SQL DDL files. These descriptions are used to compare, or make changes to other schemata.

The *Compare Database Objects* application allows DBA users to compare different "database" versions. This way, in case of an application error produced by a non-tested schema change applied in the database, the DBA can produce all changes a-posterior and find the cause of the application failure.

The *Synchronization Wizard* of CMP supports the user in modifying an item *target* to match another item *source*. The *Synchronization Wizard* needs a comparison of the *target* and *source* items, so it works after the *Compare Database Objects* application. The *Synchronization Wizard* orders the "transformation" steps, in order to produce the *target* item. This is, for example, to make sure that the foreign keys will be applied after the primary keys. Besides that, the *Synchronization Wizard* can delete items. This happens, when there is no *source* item. Moreover, if there is no *target* item, the *Synchronization Wizard* initially creates and then synchronizes a *new target* item with the *source* one. Finally, using the *Synchronization Wizard*, the user may keep or undo the changes made to a *target* item.

Another module that works similar to the *Synchronization Wizard* is the *DB Propagate* application of CMP, which allows the user to select one or more object definitions and reproduce them in one or more *target* schemata.

Two other applications of CMP are: *DB Quick Change*, and, *DB Alter*. The *DB Quick Change* application helps the user in making *one* change to a *single* database item. The *DB Alter* application helps the user in making one or more changes to one, or more database items (in comparison to the *Synchronization Wizard*, here there is no need of any preceding comparison).

Finally, the *Plan Editor* of CMP lets the user perform a single change plan on one or more databases, that he may keep or undo. The *Plan Editor* can perform a wider variety of changes, compared to those that *Synchronization Wizard*, *DB Alter*, *DB Quick Change*, and *DB Propagate* can perform. The *Plan Editor* allows the creation of a change plan that serves as a container for change requests (directives, scoped directives, exemplars, and modified exemplars), generates scripts for those change requests and executes them on one or more databases.

**IBM - DB2.** IBM DB2 provides a mechanism that checks the type of the schema changes [18] that the users want to perform in system-period temporal tables. A system-period temporal table is a table that maintains historical versions of its rows. A system-period temporal table uses columns that capture the begin and end times when the data in a row is valid and preserve historical versions of each table row whenever updates or deletes occur. In this way, queries have access to both current data, i.e., data with a valid current time, as well data from the past. Finally, DB2 offers the DB2 Object Comparison Tool [19]. It is used for identifying structural differences between two or more DB2 catalogs, DDL, or version files (even between objects with different names). Moreover, it is able to generate a list of changes in order to transform the *target* comparator

into a new schema, described by the *source* comparator. Finally, it is capable to undo changes that were performed and committed in a version file, so as to restore it to a given previous version.

Temporal tables prohibit changes that result in loss of data. These changes can be produced by commands like DROP COLUMN, ADD COLUMN, and, ALTER COLUMN. All the changes, applicable to temporal tables, can be propagated back to the history of the schema, with only two exceptions, the renaming of a table and the renaming of an index.

**Microsoft - SQL Server and Visual Studio.** Change management support for Microsoft SQL Server comes with the SQL Server Management Studio [20] (SSMS). SSMS allows the user to browse, select, and manage any of the database objects (e.g., create a new database, alter an existing database schema, etc.) as well as visually examine and analyze query plans and optimize the database performance. SSMS provides data import export capabilities, as well as data generation features, so that users can perform validation tests on queries. Regarding the evolution point of view, it is capable of comparing two different database instances and returning their structural differences. The tool may also provide information on DDL operations that occurred, through the reports of schema changes. An example of such a report from [21] is displayed in Table 1.

**Table 1.** SSMS report

| database name | start time | login name | user name | application name | ddl operation | object | type desc |
|---|---|---|---|---|---|---|---|
| msdb | 2015-08-27 14:08:40.460 | sa | sa | SSMS - Query | CREATE | dbo.DDL_History | USER_TABLE |
| TestDB | 2015-08-26 11:32:19.703 | sa | sa | SSMS | ALTER | dbo.SampleData | USER_TABLE |

Another set of tools that Microsoft offers for the validation of SQL code is the SQL Server Data Tools [22] (SSDT). SSDT follows a project-based approach for the database schema and SQL source code that is embedded in the applications. A developer can use SSDT to locally check and debug SQL code (by using breakpoints in his SQL code).

Another tool that comes from Microsoft as an extension to Visual Studio is MoDEF [23]. MoDEF uses the model-view-controller idea for the database schema manipulation. In MoDEF, the user defines classes that represent the columns of a table in a relational database. The classes are mapped to relational tables that are created in the database via select-project queries. In MoDEF, the changes of the client model are translated to incremental changes as an upgrade script for the database. Moreover, MoDEF annotates the upgrade script with comments, to inform the DBA for the changes that are going to happen in the database schema.

### 3.2 Open Source or Academic Tools

**Django.** Likewise to MoDEF, Django [24] also uses the model-view-controller idea for the database schema manipulation. Regarding evolution, Django uses an automatic way to identify which columns were added or deleted from the tables between two versions of code and migrate these changes to the database schema. Django identifies the changes in the attributes of a class and then produces the appropriate SQL code that performs the changes to the underlying database schema.

**South.** South [25] is a tool operating on top of Django, identifying the changes in the Django's models and providing automatic migrations to match the changes. South supports five database backends (PostgreSQL, MySQL, SQLite, Microsoft SQL Server, and, Oracle), while Django officially supports four (PostgreSQL, MySQL, SQLite, and, Oracle). South also supports another five backends(SAP SQL Anywhere, IBM DB2, Microsoft SQL Server, Firebird, and, ODBC) through unofficial third party database connectors.

In South, one can express dependencies of table versions so as to have the correct execution order of migration steps and void inconsistencies. For example, in a case where a foreign key references a column that is not yet a key, this kind of problem can be identified and avoided.

The Autodetector part of South can extend the migrations that Django offers. Specifically, South can automatically identify the following schema modifications: model creation and deletion (create/drop a table), field changes (type change of columns) and unique changes, while Django can only identify the addition or deletion of columns.

**Hecate.** Hecate [26] is a tool that parses the DDL files of a project and compares the database schemata between versions. Hecate also exports the transitions between two versions, describing the additions and deletions that occurred between the versions (renames are treated as deletions followed by additions). Hecate also provides measures such as size and growth of the schema versions.

**Hecataeus.** Hecataeus [27] is a what-if analysis tool that facilitates the visualization and impact analysis of data-intensive software ecosystems. As these ecosystems include software modules that encompass queries accessing an underlying database, the tool represents the database schema along with its dependent views and queries as a uniform directed graph. The tool visualizes the entire ecosystem in a single representation and allows zooming in and out its parts. Most related to the topic of this survey, the tool enables the user to create hypothetical evolution events and examine their impact over the overall graph. Hecataeus does not simply flood the event over the underlying graph; it also allows users as to define "veto" rules that block the further propagation of an evolutionary event (e.g., because a developer is adamant in keeping the exact

structure of a table employed by one of her applications). Hecataeus also rewrites the graph, after the application of the event so that both the syntactical and the semantic correctness of the affected queries and views are retained.

## 4 Techniques for Managing Database and View Evolution

In this section, we discuss the impact of changes in a database schema to the applications that are related to that schema. Given a set of scripts, the methods proposed in this part of the literature identify how database and software modules are affected by changes that occur at the database level. Techniques for query rewriting are also discussed. Closely to this topic is the topic of view adaptation: how must the definition (and the extent, in case of materialization) of a view adapt whenever the schema of its underlying relations changes?

### 4.1 Impact Assessment of Database Evolution

The problem of impact assessment for evolving databases has two facets: (a) the identification of the parts of applications that are affected by a change, and, (b) the automation of the rewritting of the affected queries, once they have been identified. In this subsection, we organize the discussion of related efforts in a way that reflects both the chronological and the thematic dimension of how research has unfolded. A summary of the different methods is given at the end of the subsection.

**Early Attempts Towards Facilitating Impact Assessment.** Maule, Emmerich and Rosenblum [28] propose a technique for the identification of the impact of relational database schema changes upon object-oriented applications. In order to avoid a high computational cost, the proposed technique uses slicing, so as to reduce the size of the program that is needed to be analyzed. At a first step, the authors use a prototype slicing implementation that helps them identify the database *queries* of the program. Then, with a data-flow analysis algorithm, the authors estimate the possible runtime values for the parameters of the query. Finally, the authors use an impact assessment tool, Crocopat, coming with a reasoning language (RML) to describe the impacts of a potential change to the stored data of the previous step. Depending on the type of change, a different RML program is run, and this eventually isolates the *lines of code of the program* that are related to the queries affected by the change. The authors evaluated their approach on a C# CMS project of 127000 lines of code, and a primary database schema of up to 101 tables, with 615 columns and 568 stored procedures. The experiments showed that the method needed about 2 min for each execution, where they found that there were no false negatives. On the other hand, there were false positives in the results, meaning that the tool was able to find all the lines of code that were affected, leaving none out, but also falsely reported that some lines of code would be affected, whilst this was not really happening.

**Architecture Graphs.** Papastefanatos et al. [29–31] introduced the idea of dealing with *both* the database and the application code in uniform way. The results of this line of research are grouped in the areas of (a) modeling, (b) change impact analysis, and (c) metrics for data intensive ecosystems (data intensive ecosystems are conglomerations of data repositories and the applications that depend on them for their operations). This line of work has been facilitated by the Hecataeus tool (see [29,32]).

Concerning the modeling and the impact analysis parts, in [29], the authors proposed the use of the *Architecture Graph* for the modeling of data intensive ecosystems. The *Architecture Graph* is a directed graph where the nodes represent the entities of the ecosystem (relations, attributes, conditions, queries, views, group by clauses, etc.), while the edges represent the relationships of these entities (schema relationships, operand relationships, map-select relationships, from relationships, where relationships, group by relationships, etc.). In the same paper, the authors proposed an algorithm for the propagation of the changes of one entity to other related entities, using a *status* indicator of whether the imminent change is accepted, blocked or if the user of the tool should be asked.

In [30], the authors proposed an extension for the SQL query language, that introduced policies for the changes in the database schema. The users could define in the declaration of their database schema whether a change should be accepted, blocked or if the user should be prompted. In this work, the policies were defined over: (a) the database schema universally, (b) the *high level modules* (relations, views and queries) of the database schema, and, (c) the remaining entities of the database, such as attributes, constraints and conditions.

Regarding the metrics part, a first attempt to the problem was made by Papastefanatos et al., on ways to predict the maintenance effort and the assessment of the design of ETL flows of data warehouses under the prism of evolution in [31]. In [33], the same authors used a real world evolution scenario, which used the evolution of the Greek public sector's data warehouse maintaining information for farming and agricultural statistics. The experimental analysis of the authors is based in a six-month monitoring of seven real-world ETL scenarios that process the data of the statistical surveys. The Architecture Graph of the system was used as a provider of graph metrics. The findings of the study indicate that schema size and module complexity are important factors for the vulnerability of an ETL flow to changes.

In a later work [34], Manousis et al., redefine the model of the *Architecture Graph*. The paper extends the previous model by requiring the *high level modules* of the graph to include *input* and *output* schemata, in order to obtain an isolation layer that leads to the simplification of the policy language. The method is based on the annotation of modules with *policies* that regulate the propagation of events in the Architecture Graph; thus, a module can either block a change or adapt to it, depending on its policy. The method for impact assessment includes three steps that: (a) assess the impact of a change, (b) identify policy conflicts

from different modules on the same change event, and (c) rewrite the modules to adapt to the change. It is noteworthy that simply flooding the evolution event over the *Architecture Graph* in order to assess the impact and perform rewrittings, is simply not enough, as different nodes can react with controversial policies to the same event. Thus, the three stages are necessary, with the middle one determining conflicts and a "cloning" method, for affect paths on the graph, in order to service conflicting requirements, whenever possible.

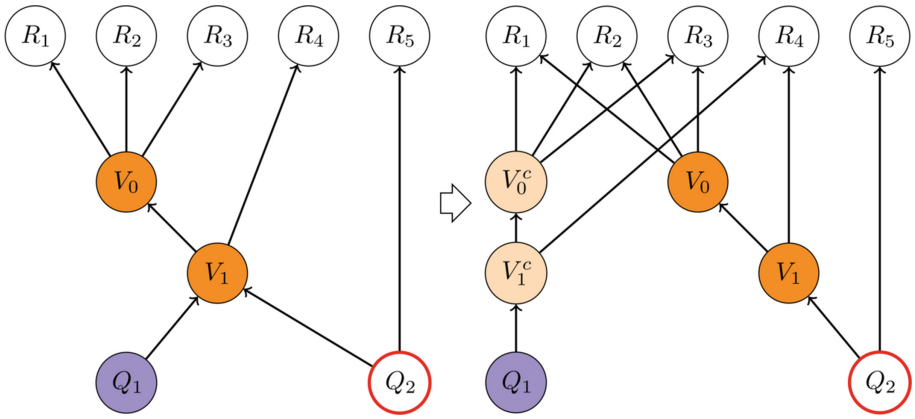


**Fig. 1.** A example of a rewrite process when the policies of $Q_1$ and $Q_2$ queries are conflicting [35].

In Fig. 1, we depict a situation that exemplifies the above. In the Architecture Graph that is displayed in the left part of Fig. 1, a change happens in view $V_0$ and affects the view $V_1$, which, in turn, affects the two queries $Q_1$ and $Q_2$ of the example. The first query ($Q_1$) accepts the change, whereas the second one ($Q_2$) blocks it. This means that $Q_2$ wants to retain its semantics and be defined over the old versions of the views of the Architecture Graph. Therefore, the query that accepted the change will get a new path, composed of "cloned", modified versions of the involved views that abide by the change (depicted in light color in the left part of the figure and annotated with a superscript $c$), whereas the original views and their path towards $Q_2$ retain their previous definition (i.e., they decline the change).

**Schema Modification Operators.** In this section, we review a work that produces –when it is possible– valid query rewritings of old queries over a new database schema, as if the evolution step of the database schema never happened. This way, the results that the user receives, after the execution of the rewritten query, are semantically correct.

An approach that supports the ecosystem idea, to a certain extent, is [36]. In this approach, the authors propose a method that rewrites queries whenever one

of their underlying relations changes with the goal of retaining the same query result as if the evolution event never happened, using *Schema Modification Operators* (SMOs). The Schema Modification Operators that PRISM/PRISM++ tool uses are:

- `CREATE TABLE` $R(a, b, c)$
- `DROP TABLE` $R$
- `RENAME TABLE` $R$ `INTO` $T$
- `COPY TABLE` $R$ `INTO` $T$
- `MERGE TABLE` $R$, $S$ `INTO` $T$
- `PARTITION TABLE` $R$ `INTO` $S$ `WITH condition`, $T$
- `DECOMPOSE TABLE` $R$ `INTO` $S(a, b)$ $T(a, c)$
- `JOIN TABLE` $R$, $S$ `INTO` $T$ `WHERE condition`
- `ADD COLUMN` $d$ [`AS constant` | `function`$(a, b, c)$] `INTO` $R$
- `DROP COLUMN` $r$ `FROM` $R$
- `RENAME COLUMN` $b$ `IN` $R$ `TO` $d$

The $R$, $S$, and $T$ variables represent relations. The $a$, $b$, $c$, $d$, and $r$ variables represent attributes. The `constant` variable stands for a fixed value, while the `function` is used in `ADD COLUMN` in order to express simple tasks as data type and semantic conversions are. Besides the schema modification operators, PRISM/PRISM++ uses the integrity constraints modification operators ICMO and policies (which will be described later on) for this kind of rewrites. The ICMOs are:

- `ALTER TABLE` $R$ `ADD PRIMARY KEY` $pk1(a, b)$ <`policy`>
- `ALTER TABLE` $R$ `ADD FOREIGN KEY` $fk1(c, d)$ `REFERENCES` $T(a, b)$ <`policy`>
- `ALTER TABLE` $R$ `ADD VALUE CONSTRAINT` $vc1(c, d)$ AS $R.e$="0" <`policy`>
- `ALTER TABLE` $R$ `DROP PRIMARY KEY` $pk1$
- `ALTER TABLE` $R$ `DROP FOREIGN KEY` $fk1$
- `ALTER TABLE` $R$ `DROP VALUE CONSTRAINT` $vc1$

The $R$ and $T$ variables represent relations. The $a$, $b$, $c$, $d$, and $e$ variables represents attributes. The $pk1$ represents the primary key of the preceding relation. The $fk1$ represents the foreign key of a relation. Finally, the $vc1$ represents a value constraint. The ICMOs have, also, a <`policy`> placeholder, where the policy can be one of the following:

1. `CHECK`, where the PRISM/PRISM++ tool verifies that the current database satisfies the constraint, otherwise the ICMO is rolled back,
2. `ENFORCE`, where the tool removes all the data that violate the constraint, and,
3. `IGNORE`, where the tool ignores if there exist tuples that violate the constraint or not, but informs the user about this.

When the `ENFORCE` policy is used and tuples have to be removed, the tool creates a new database schema and inserts all the violating tuples in order to help the DBA carry out inconsistency resolution actions.

Regarding the rewrite process of queries through SMOs, the Chase &Backchase algorithm uses as input the SMOs and a query that is to be rewritten. The algorithm rewrites the query through an inversion step of the SMO's (for example, the inversion of a JOIN is the DECOMPOSITION), in order to retain the query's results unchanged, independently of the underlying schema. This way, the resulting tuples of the query will be the same as if the database schema never changed. The rewrite process of queries through ICMOs is done with the help of policies.

So, the steps that describe the algorithm of the rewriting that the authors proposed, are:

1. Get the SMOs from the DBA
2. Inverse the SMOs, in order to guarantee –if it is possible– the semantic correctness of the new query
3. Rewrite the query and validate its output.

The authors also describe a rewrite process of updates statement queries ("UPDATE table SET...") through SMOs and ICMOs, based in the ideas described in the previous paragraph. If the rewrite is through SMOs, the UpdateRewrite algorithm tries to invert the evolution step, while if the rewrite is through ICMOs, the policies ask the tool to check the tuples of the database and either guarantee or inform the user about the contents of the database.

To improve their rewrite time the authors try to minimize the input of the Chase &Backchase algorithm, by removing from the input all the mappings and constraints that are not related with the evolution step. Moreover, the proposed method uses only the version of the relation in which the query was written, leaving all the previous modifications out, as they are unrelated to the query. This is the backchase optimizer technique that the authors proposed, which produced bigger execution times in the chase and backchase phase of higher connected schemata because of the foreign keys that lead to higher input in chase phase, in the experiments that were conducted. In order to achieve even better execution time, the authors propose the use of a caching technique, since from the observations they made on their datasets, they noticed that there is a number of common query/update templates, which is parametrized and reused multiple times. These patterns are:

*Join pattern type 1.* In this pattern, a new table is created to host joined data from the desired column of two or more tables and migrates the data from the old tables to the new one.

*Join pattern type 2.* In this pattern, the data of a column are moved from the source table to the destination table.

*Decompose pattern.* In this pattern, a table is decomposed to two new tables. In order to be correct, both tables should have the key of the table.

*Partition pattern.* In this pattern, a part of the data of a table is moved into a new table and deleted from the original one.

*Merge pattern.* In this pattern, all the tuples of a table are moved into another table.
*Copy pattern.* In this pattern, an existing table is cloned.

The authors validated the PRISM/PRISM++ tool using the Ensembl project, including 412 schema versions, and the Mediawiki project, which is part of the Wikipedia project and had 323 schema versions. The authors used 120 SQL statements (queries and updates) from those two projects, tested them against SMO and ICMO operators and their tool found a correct rewriting, whenever one existed.

In a later work [6], the authors provide an extended description of the tool that performs the rewrites of the queries (PRISM/PRISM++) and its capabilities. Moreover, the authors introduce two other tools of which the first one collects and provides statistics on database schema changes and the other derives equivalent sequences of (SMOs) from the migration scripts that were used for the schema changes.

**Summary.** In Table 2 we summarize the problems and the solutions of the works that were presented earlier. The first two works are dealing with the impact analysis problem, which is to identify which parts of the code is affected by a change, and the other two works are dealing with the rewriting of the code in order to obtain or hide the schema changes.

### 4.2 Views: Rewriting Views in the Context of Evolution

A view is a query expression, stored in the database dictionary, which can be queried again, just as if it was a regular relation of the database. A view, thus, retains a dual nature: on the one hand, it is inherently a query expression; yet, on the other hand, it can also be treated as a relation. A *virtual view* operates as a macro: whenever used in a query expression, the query processor incorporates its definition in the query expression and the query is executed afterwards. *Materialized views* are a special category of views, that persistently store the results of the query in a persistent table of the DBMS. In this section, we survey research efforts that handle two problems. First, we start with the effect that a materialized view redefinition has on the maintenance of the view contents: the expression defining the view is altered and the stored contents of the view have to be adjusted to fit the new definition (ideally, without having to fully recompute the contents of the view from scratch). Second, we survey efforts pertaining to how views should be adapted when the schema of their defining tables evolves (also known as the "view adaptation" problem). A summary table concludes this subsection.

In [37], Mohania deals with the problem of maintaining the extent of a materialized view that is under redefinition, by proposing methods that try to avoid the full re-computation of the view. The author uses *expression trees*, which are binary trees, the leaf nodes represent base relations that are used for defining

**Table 2.** Summary table for Sect. 4.1

| Works | Problem | Input | Output | Method |
|---|---|---|---|---|
| [28] | Impact analysis of an imminent schema change in OO apps | DB schema and source code; an imminent change | The lines of code that are affected by the DB schema change | Slicing technique to identify the DB related lines of C# code, and estimation of values so as to further slice the C# code. |
| [29,30] | Impact analysis of an imminent schema change | DB schema and application's queries abstracted as *Architecture Graph*; policies for of the nodes; an imminent change | Annotation of affected nodes with a status indication. | Language for node annotation. Propagation of a change, based on the node's *policy* for the change. |
| [34] | Restructuring of DB schema and app queries due to a schema change | DB schema and application's queries abstracted as *Architecture Graph*; policies for of the nodes; an imminent change | Rewritten *Architecture Graph* acording to the policies | Rewrite via cloning the queries that want to acquire the change and leave intact the ones that block the change. |
| [6,36] | Rewritting of app queries due to schema change | SMOs and ICMOs of the modification, and queries that use the modified table/view | Rewritten queries returning the same result as if the change has never happened. | The 1 hop away queries are rewritten as if the schema change never happened, using the *Chase &Backchase* algorithm |

the view, while the rest of the nodes contain binary relational algebraic operators. Unary operators such as selection and projection are associated with the edges of the tree. In a nutshell, the author proposes that making use of these expression trees, it is easy to find common subexpressions between the new and old view statements and thus, if applicable, make use of the old view to get the desired results of the redefined view, without recomputing the new definition. Due to its structure, the tree allows to avoid interfering with the result of the view computation: (a) the height of the trees is no more than two levels, and, (b) a change is either a change to a unary operator associated with the edge of the tree, or a change to a binary node. This way, when the change is made at the root node, then the expression corresponding to the right hand child in the tree has to be evaluated only, while when the change is made at level `d`=1, the view re-computation becomes a view maintenance problem. Finally, when the change

is made at any other node, it is only the intermediate results of the nodes that have to be maintained.

Gupta, Mumick, Rao and Ross [38] provide a technique that redefines a materialized view and adapts its extent, as a sequence of primitive local changes in the view definition, in order to avoid a full re-computation. Moreover, on more complex adaptations –when multiple simultaneous changes occur on a view– the local changes are pipelined in order to avoid intermediate creations of results of the materialized view. The following changes are supported as primitive local changes to view definitions:

1. Addition or deletion of an attribute in the `SELECT` clause.
2. Addition, deletion, or modification of a predicate in the `WHERE` clause (with and without aggregation).
3. Addition or deletion of a join operand (in the `FROM` clause), with associated equijoin predicates and attributes in the `SELECT` clause.
4. Addition or deletion of an attribute from the `GROUP BY` list.
5. Addition or deletion of an aggregate function to a `GROUP BY` view.
6. Addition, deletion or modification of a predicate in the `HAVING` clause. Addition of the first predicate of deletion of the last predicate corresponds to addition and deletion of the `HAVING` clause itself.
7. Addition of deletion of an operand to the `UNION` and `EXCEPT` operators.
8. Addition or deletion of the `DISTINCT` operator.

Concerning the problem of adapting a view definition to changes in the relations that define it, Nica, Lee and Rundensteiner [39] propose a method that makes legal rewritings of views affected by changes. The authors primarily deal with the case of relation deletion which (under their point of view) is the most difficult change of a database schema, since the addition of a relation, the addition of an attribute, the rename of a relation and the rename of an attribute can be handled in a straightforward way (the attribute deletion, according to the authors, is a simplified version of the relation deletion). To attain this goal one should find valid replacements for the affected components of the existing view, so, in order to achieve that, the authors of [39] keep a Meta-Knowledge Base on the join constraints of the database schema. This Meta-Knowledge Base (MKB) is modeled as a hyper-graph that keeps meta-information about attributes and their join equivalence attributes on other tables. The proposed algorithm, has as input the following: (a) a change in a relation, (b) MKB entities, and, (c) new MKB entities. Assuming that valid replacements exist, the system can automatically rewrite the view via a number of joins and provide the same output as if there was no deletion. The main steps of the algorithm are: (a) find all entities that are affected for Old MKB to became New MKB, (b) mark these entities and for each one of them find a replacement from Old MKB, using join equivalences, and, (c) rewrite the view over these replacements. Interestingly, the authors accompany their method with a language called E-SQL that annotates parts of a view (exported attributes, underlying relations and filters) with respect to two characteristics: (a) their dispensability (i.e., if the part can be

**Table 3.** Summary table for Sect. 4.2

| Work | Problem | Input | Output | Method |
|---|---|---|---|---|
| [37] | Maintenance of redefined materialized views | Definition and redefinition of a materialized view | Recomputed content of the redefined view | Use of expression trees that identify common subexpressions between the input and output of their method, thus helping to avoid the full re-computation of a materialized view |
| [38] | Maintenance of redefined materialized views | Definition and redefinition of a materialized view | Recomputed content of the redefined view | The redefinition takes place as a sequence of primitive local changes (in complex adaptations this sequence is pipelined to avoid temporal results). |
| [39] | View adaptation on column deletion | Hypergraph that contains the join constraints of the DB schema | Valid replacement of column that is to be deleted | Search in the hypergraph (named MKB) for a replacement of the column that is to be deleted, and replace that column in the view with the replacement |

removed from the view definition completely) and (b) their replaceability with an another equivalent part.

**Summary.** In Table 3 we summarize the problems and the solutions of the works that were presented earlier. The first two works refer to the problem of the recomputation of the contents of a materialized view, after a redefinition of the view. The other work refers to the problem of view adaptation on a column deletion in the source tables, via a replacement.

## 5 Techniques for Managing Data Warehouse Evolution

A research area where the problem of evolution has been investigated for many years is the area of data warehouses. In this section, we concentrate on works related on evolution of both schema and data modifications in the context of data warehouses, and we review methods and tools that help on the adaptation of those changes. We also refer the reader to two excellent surveys on the issue, specifically, [40,41].

### 5.1 Data Warehouses and Views

At the beginning of data warehousing, people tended to believe that data warehouses were collections of materialized views, defined over sources. In this case,

evolution is mostly an issue of adapting the views definitions whenever sources change.

Bellahsene, in two articles, [42,43], proposed a language extension to annotate views with a HIDE clause that works oppositely to SELECT (i.e., the idea is to project all attributes except for the hidden ones and an ADD ATTRIBUTE clause to equip views with attributes not present in the sources (e.g., timestamps or calculations). Then, in the presence of an event that changes the schema of a data warehouse source (specifically, the events covered are attribute/relation addition and deletion), the methods proposed by the author for the adaptation of the warehouse handle the view rematerialization problems i.e., how to recompute the materialized extent via SQL commands. The author also proposes a cost model to estimate the cost of alternative options.

In [44], the author proposes an approach on data warehouse evolution based on a meta-model, that provides complementary metadata that track the history of changes (in detail, changes that are related to data warehouse views) and provide a set of consistency rules to enforce when a quality factor (actual measurement of a quality value) has to be re-evaluated.

### 5.2 Evolution of Multidimensional Models

Multidimensional models are tailored to treat the data warehouse as a collection of *cubes* and *dimensions*. Cubes represent clean, undisputed facts that are to be loaded from the sources, cleaned and transformed, and eventually queried by the client application, Cubes are defined over unambiguous, consolidated dimensions that uniquely and commonly define the context of the facts. Dimensions comprise *levels*, which form a hierarchy of degrees of detail according to which we can perform the grouping of facts. For example, the Time dimension can include the levels (1) Day, that can be rolled up to either (2a) Week or (2b) Month, both of which can be rolled up to level (3) Year. Each level comes with a domain of values that belong to it. The values of different levels are interrelated via rollup functions (e.g., 1/1/2015 can be rolled up to value 1/2015 at the Month level). As levels construct a hierarchy that typically takes the form of a lattice, evolution is mainly concerned with changing (i) the nodes of the lattice, or (ii) their relationship, or (iii) the values of the levels and their interrelationship. The problem that arises, then, is: how do we adapt our cubes (in their multidimensional form and possibly their relational representation) when the structure of their dimensions changes? The works surveyed in this subsection address this problem. A table at the end of the subsection summarizes the problems addressed and the solutions that are given.

The authors of [45] present a formal framework, based on a formal conceptual description of an evolution algebra, to describe evolutions of multi-dimensional schemata and their effects on the schema and on the instances. In [45], the authors propose a methodology that supports an automatic adaptation of the multi-dimensional schema and instances, independently of a given implementation. The main objectives of the proposed framework are: (i) the automatic adaptation of instances, (ii) the support for atomic and complex operations, (iii) the

definition of semantics of evolution operations, (iv) the notification mechanism for upcoming changes, (v) the concurrent operation and atomicity of evolution operations, (vi) the set of strategies for the scheduling of effects and (vii) the support of the design and maintenance cycle.

The authors provide a minimal set of atomic evolution operations, which they use in order to present more complex operations. These operations are: (i) insert level, (ii) delete level, (iii) insert attribute, (iv) delete attribute, (v) connect attribute to dimension level, (vi) disconnect attribute from dimension level, (vii) connect attribute to fact, (viii) disconnect attribute from fact, (ix) insert classification relationship, (x) delete classification relationship, (xi) insert fact, (xii) delete fact, (xiii) insert dimension into fact, and, finally, (xiv) delete dimension.

In [46], the authors suggest a set of primitive dimension update operators that address the problems of: (i) adding a dimension level, above (generalize) or below (specialize) an existing level, (ii) deleting a level, (iii) adding or deleting a value from a level (add/delete instance), or (iv) adding (relate) or removing edges between parallel levels (unrelate). In [46], the authors also suggest another set of complex operators, that intend to capture common sequences of changes in instances of a dimension and encapsulate them in a single operation. The set of those operators consists of: (i) reclassify (used, for example, when new regions are assigned to salespersons as a result of marketing decisions of a company), (ii) split (used, for example, when a region is divided into more regions and more salespersons must be assigned to those regions due to the population density), (iii) merge (the opposite of split), and, (iv) update (used, for example, when a brand name for a set of items changes but the corporation as well as the set of products related to the brand remain unchanged).

The mappings that the authors propose, for the transitions from the multidimensional to the relational model, support both the de-normalized and normalized relational representations. In the de-normalized approach, the idea is to build a single table containing all the roll-ups in the dimension while in the normalized approach, the idea is to build a table for each direct roll-up in the dimension.

Finally, in the experiments that the authors conducted, they found that the structural update operators in the de-normalized representation are more expensive. The instance update operators in the normalized representation are more expensive because of the joins that have to be performed, whilst both representations are equally good for the operators that compute the net effect of updates.

In a later work, the authors of [47] suggest a set of operators which encapsulate common sequences of primitive dimension updates and define two mappings from the multidimensional to the relational model, suggesting a solution on the problem of multidimensional database adaptation.

The effects of evolution to alternative relational logical designs is explored in [48]. Specifically, the authors explore the impact of changes to both star and snowflake schemata. The changes covered include (i) the addition of deletion

of attributes to levels, (ii) the addition/deletion of dimension levels, (iii) the addition/deletion of measures, and (iv) the addition/deletion of dimensions into fact tables. A notable, albeit expected, result is that comparison of the effect of changes to the two alternative structures, reveals that the simplest one, star schema, is more immune to change than the more complicated one.

**Summary.** In Table 4 we summarize the problems and the solutions of the research efforts that were presented earlier. The first two lines of work refer to the evolution of multidimensional database schemata and the adaptation of its contents, and the final effort refers to a comparison of the logical design between star and snowflake alternatives.

### 5.3 Multiversion Querying over Data Warehouses

Once the research community had obtained a basic understanding of how multidimensional schemata can be restructured, the next question that followed was: "what if we keep track of the history of all the versions of a data warehouse schema as it evolves?" Then, we can ask queries that span several versions

**Table 4.** Summary table for Sect. 5.2

| Works | Problem | Input | Output | Method |
|---|---|---|---|---|
| [45] | Multidimensional database adaptation | MD schema; Changes of schemata | New schema and instances | Automatic adaptation of multi-dimensional schema and instances through simple and complex operators of an evolution algebra |
| [46,47] | Multidimensional database adaptation | MD schema; Changes of schemata | Normalized or de-normalized new (RDBMS) schema | Use of primitive dimension update operators and complex operators that map the multidimensional schemata to RDBMS schemata |
| [48] | Evolution of alternative relational logical designs | Changes of schemata | Comparison of logical designs to changes | Perform the changes to both star and snowflake designs |

having different structure, also known as *multiversion queries*. The essence of multi-version queries involves transforming the data of previous versions (that obey a previous structure) to the current version of the structure of the data warehouse, in order to allow their uniform querying with the current data.

In this section, we discuss the adaptation of multiversion data warehouses [49], the use of data mining techniques in order to detect structural changes in data warehouses [50–52], and, the use of graph representations (directed graphs) [53], in order to achieve correct cross version queries. We summarize problems and solutions in a table at the end of the subsection.

Eder and Koncilia [52] propose a multidimensional data model that allows the registration of temporal versions of dimension data in data warehouses. Mappings are provided to translate data between different temporal versions of instances of dimensions. This way, the system can answer correctly queries that span in periods where dimension data have changed. The paper makes no assumption on dimension levels, so when referring to a dimension, the paper implies a flat structure with a single domain. The mappings are described as transformation matrices. Each matrix is a mapping of data from version $\mathtt{V}_i$ to version $\mathtt{V}_{i+1}$ for a dimension `D`. Assume, for example a 2-dimensional cube, including dimensions $A$ and $B$ with domains $\{a_1, a_2\}$ and $\{b_1, b_2\}$ respectively. Assume that in a subsequent version: (i) $a_1$ is split to $a_1^1$ and $a_1^2$ and (ii) $b_1$ and $b_2$ are merged into a single value $b$. Then, there is a transformation matrix for dimension $A$, with one row for each old value $\{a_1, a_2\}$ and one column for each new value $\{a_1^1, a_1^2, a_2\}$ expressing how the previous values relate to the new ones. For example, one might say that $a_1^1$ takes 30 % of $a_1$ and $a_1^2$ takes the other 70 %. The respective matrix is there for dimension $B$. Then, by multiplying any cube with $A$ and $B$ as dimensions with the respective transformation matrices, we can transform an old cube defined over $\{a_1, a_2\} \times \{b_1, b_2\}$ to a new cube defined over $\{a_1^1, a_1^2, a_2\} \times \{b\}$.

So at the end, the resulting factual cube maps the data of the previous version to the dimension values of the current version; this way, both the current and the previous version can be presented uniformly to the user.

Eder, Koncilia and Mitsche [50] propose the use of data mining techniques for the detection of structural changes in data warehouses, in order to achieve correct results in multi-period data analysis OLAP queries. Making use of three basic operations (INSERT, UPDATE and DELETE), the authors are able to represent more complex operations such as: SPLIT, MERGE, CHANGE, MOVE, NEW-MEMBER, and DELETE-MEMBER. The authors propose several data mining techniques that detect which is the schema attribute that changed. In the experiments that were conducted, the authors observed that the quality of the results of the different methods depends on the quality and the volatility of the original data.

The same authors continue their previous work on data mining techniques for detection of changes in OLAP queries in [51]. Since their previous approach was incapable of detecting some variety of changes, the authors propose data mining techniques in form of multidimensional outlier detection to discover unexpected

deviations in the fact data, which suggests that changes occurred in dimension data. By fixing a dimension member they get a simple two-dimensional matrix where the one axis is the excluded dimension member. From that matrix, a simple deviation matrix with relative differences is computed. In this deviation matrix, the results are normalized to get the probability of a structural change that might have occurred. The authors propose the 10 % as a probability threshold for the change to have occurred. From the conducted experiments, the authors found that this method analyzes the data in more detail and gives a better quality of the detected structural changes.

Some years later, Golfarelli et al. [53] propose a representation of data warehouse schemata as graphs. The proposed graph represents a data warehouse schema, in which the nodes are: (i) the fact tables of the data warehouse, and (ii) the attributes of fact tables (including properties and measures), while the edges represent simple functional dependencies defined over the nodes of the schema. The authors also define an algebra of schema graph modifications that are used to create new schema versions and discuss of how cross-version queries can be answered with the help of augmented data warehouse schemata. The authors finally show how a history of versions for data warehouse schemata is managed.

Since the authors' approach is based on a graph, the schema modification algebra uses four simple schema modification operations ($M$): (i) $\texttt{Add}_F$ that adds an arc involving existing attributes, (ii) $\texttt{Del}_F$ that deletes an existing arc, (iii) $\texttt{Add}_A$ that adds a new attribute –directly connected by an arc to its fact node– and (iv) $\texttt{Del}_A$ that deletes an existing attribute. Besides those simple operators, the authors define the $\texttt{New}(S, M)$ operator that describes the creation of a new schema, based on the existing schema $S$ when a simple schema modification $M$ is applied.

The authors introduce augmented schemata to serve multiversion queries. Each previous version of the data warehouse schema is accompanied by an augmented schema whose purpose is to translate the old data under the old schema to the current version of the schema. To this end, the augmented schema keeps track of every new attribute (say $A$), or new functional dependency (say $f$). In order to translate the old data to the new version of the schema, the system might have to: (i) estimate values for $A$, (ii) disaggregate or aggregate measure values depending on the change of granularity, (iii) compute values for $A$, (iv) add values for $A$, or, (v) check if $f$ holds.

The set of versions of the schemata is described by a triple $(S, S^{AUG}, t)$, where $S$ is a version, $S^{AUG}$ is the related augmented schema and $t$ is the start of the validity interval of $S$. This way, the history of the versions of the data warehouse can be described as a sequence of changes over changes, starting from the initial schema of the history: $H = S_0, S_0^{AUG}, t_0$. Since every previous version is accompanied by an augmented schema that transforms it to the current one, it is possible to pose a query that spans different versions and translate the data of the previous versions to a representation obeying the current schema, as explained above.

Practically around the same time, Wrembel and Bebel [49] deal both with cross-version querying and with the problems that appear when changes take place at the external data sources (EDS) of a data warehouse. Those problems can be related to a multi-version data warehouse which is composed of a sequence of persistent versions that describe the schema and data for a given period of time. The authors approach has a meta-data model with structures that support: (i) the monitoring of the external data sources on content and structural changes, (ii) the automated generation of processes monitoring external data sources, (iii) the adaptation of a data warehouse version to a set of discovered external changes, (iv) the description of the structure of every data warehouse version and (v) the querying of multiple data warehouse versions (cross version querying), and (vi) the presentation of the output as an integrated result.

The schema change operations that the authors support are: (i) the addition of a new attribute to a dimension level table, (ii) the removal of an attribute from a dimension level table, (iii) the creation of a new fact table, (iv) the association of a fact table with a dimension table, (v) the renaming of a table, and three more operations that are applicable to snowflake schemata, (vi) the creation of a new dimension level table with a given structure, (vii) the inclusion of a parent dimension level table into its child dimension level table, and, (viii) the creation of a parent dimension level table based on its child level table.

The instance change operations that the authors have worked on, are: (i) the insertion of a new level instance into a given level, (ii) the deletion of a level instance, (iii) the change of the association of a child level instance to another parent level instance, (iv) the merge of several instances of a given level into one instance of the same level, and (v) the split of a given level instance into multiple instances of the same level.

In order to query multiple versions, the authors' method is based on a simple and elegant idea: the original query is split to a set of single version queries. Then, for each single version query, the system does a best-effort approach: if, for example, attributes are missing from the previous version, the system omits them from the single version query; the system exploits the available metadata for renames; it can even, ignore a version, if the query is a group by query and the grouping is impossible. If possible, the collected results are integrated under the intersection of attributes common to all versions (if this is the case of the query); otherwise, they are presented as a set of results, each with its own metadata.

Regarding the detection of changes in external data sources, the authors propose a method that uses wrappers (software modules responsible for data model transformations). Each wrapper is connected to a monitor (software that detects predefined events at external data sources). When an event is detected, a set of actions is generated and stored in *data warehouse update register* in order to be applied to the next data warehouse version when the data warehouse administrator calls the warehouse refresher. The events are divided into two categories: (i) structure events (which describe a modification in the schema of the data warehouse) and (ii) data events (which describe a modification in the contents of a data warehouse). For each event, an administrator defines a set of

actions to be performed in a particular data warehouse version. The actions are divided in two categories: (i) messages (which represent actions that cannot be automatically applied to a data warehouse version) and (ii) operations (for events whose outcomes can be automatically applied to a data warehouse version). Both categories of actions do not create a new data warehouse version automatically but require either the administrator to apply them *all* in an action definition of an explicitly selected version, or the actions are logged in a special structure for manual application of the ones the administrator wants to apply.

**Summary.** In Table 5 we summarize the problems and the solutions of the research efforts that were presented earlier. The first two lines of work refer to data translation between the versions of the data warehouse, while the other two efforts refer to cross-version queries.

**Table 5.** Summary table for Sect. 5.3

| Works | Problem | Input | Output | Method |
|---|---|---|---|---|
| [52] | Data translation between versions of DW | History of DW data | A derivation of the data of previous version | Transformation matrices that are mappings between the different versions of the DW |
| [50,51] | Data translation between versions of DW | History of DW data; Multi-period query | A derivation of the data answering the multi-period query | Data mining techniques that identify DW schema changes and dimension changes, using a normalized matrix |
| [53] | Data translation between versions of DW; Cross-version queries | History of DW schema | Mapping of previous schemata and data to current schema | Graphs with a simple algebra that describes schema changes and augmented schemata to translate the data from old schemata to current |
| [49] | Cross version queries & changes of external data sources | History of DW schema; Data providers; Cross version query | Answer to the cross version query | Decompose a query to queries that are correct at each schema version. For the evolution of sources, wrappers notify monitors that activate rules that respond to the change |

## 6 Prospects for Future Research

Handling data and software evolution seems to be a meta-problem that generates problems in specific subareas of computer science and data management. As such, we forecast that research problems around the evolution of data and their structure will never cease to exist.

We have covered the area of logical schema evolution in relational settings, and data warehouses in particular. The evolution of data at the instance level and at the evolution of the schema at the physical level has not been covered in this paper, although both are of great importance.

We also believe that as particular areas of data management have provided ground for research on the problem of evolution in the past (e.g., Conceptual Modeling, XML, Object-Oriented databases, etc.), the future will include research efforts in the hot topics of the day, at any given time period. For example, nowadays, we anticipate that schema-less data, or data with very flexible structures (graphs, texts, JSON objects, etc.) will offer ground for research on the management of their evolution.

Concerning the area of the impact of evolution to ecosystems, the two main areas that seem to require further investigation are: (a) the identification of the constructs that are most sensitive to evolution – ideally via metrics that assess their sensitivity to evolution, and (b) the full automation of the reaction to changes by mechanisms like self-monitoring and self-repairing.

We close with the remark that due to the huge importance and impact of evolution in the lifecycle of both data and software, the potential benefits outweight the (quite significant) risk of pursuing research of both pure scientific nature, in order to find laws and patterns of evolution, and of practical nature, via tools and methods that reduce the pain of evolution's impact.

## References

1. Roddick, J.F.: A survey of schema versioning issues for database systems. Inf. Softw. Technol. **37**(7), 383–393 (1995)
2. Hartung, M., Terwilliger, J.F., Rahm, E.: Recent advances in schema and ontology evolution. In: Bellahsene, Z., Bonifati, A., Rahm, E. (eds.) Schema Matching and Mapping. data-centric systems and applications, pp. 149–190. Springer, Heidelberg (2011)
3. Sjøberg, D.: Quantifying schema evolution. Inf. Softw. Technol. **35**(1), 35–44 (1993)
4. Curino, C., Moon, H.J., Tanca, L., Zaniolo, C.: Schema evolution in wikipedia: toward a web information system Benchmark. In: Proceedings of 10th International Conference on Enterprise Information Systems (ICEIS) (2008)
5. Curino, C.A., Moon, H.J., Zaniolo, C.: Graceful database schema evolution: the PRISM workbench. Proc. VLDB Endowment **1**, 761–772 (2008)
6. Curino, C., Moon, H.J., Deutsch, A., Zaniolo, C.: Automating the database schema evolution process. VLDB J. **22**(1), 73–98 (2013)
7. Lin, D.Y., Neamtiu, I.: Collateral evolution of applications and databases. In: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution and Software Evolution Workshops (IWPSE), pp. 31–40 (2009)

8. Wu, S., Neamtiu, I.: Schema evolution analysis for embedded databases. In: Proceedings of the 27th IEEE International Conference on Data Engineering Workshops (ICDEW), pp. 151–156 (2011)
9. Qiu, D., Li, B., Su, Z.: An empirical analysis of the co-evolution of schema and code in database applications. In: Proceedings of the 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 125–135(2013)
10. Skoulis, I., Vassiliadis, P., Zarras, A.: Open-source databases: within, outside, or beyond Lehman's laws of software evolution? In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) CAiSE 2014. LNCS, vol. 8484, pp. 379–393. Springer, Heidelberg (2014)
11. Skoulis, I., Vassiliadis, P., Zarras, A.: Growing Up with Stability: how Open-Source Relational Databases Evolve. Information Systems in press (2015)
12. Vassiliadis, P., Zarras, A.V., Skoulis, I.: How is life for a table in an evolving relational schema? birth, death and everything in between. In: Johannesson, P., Lee, M.L., Liddle, S.W., Opdahl, A.L., Pastor López, Ó. (eds.) ER 2015. LNCS, vol. 9381, pp. 453–466. Springer, Heidelberg (2015). doi:10.1007/978-3-319-25264-3_34
13. Lehman, M.M., Fernandez-Ramil, J.C.: Rules and tools for software evolution planning and management. In: Software Evolution and Feedback: Theory and Practice. Wiley (2006)
14. Belady, L.A., Lehman, M.M.: A model of large program development. IBM Syst. J. **15**(3), 225–252 (1976)
15. Herraiz, I., Rodriguez, D., Robles, G., Gonzalez-Barahona, J.M.: The evolution of the laws of software evolution: a discussion based on a systematic literature review. ACM Comput. Surv. **46**(2), 1–28 (2013)
16. Lehman, M.M., Fernandez-Ramil, J.C., Wernick, P., Perry, D.E., Turski, W.M.: Metrics and laws of software evolution - the nineties view. In: Proceedings of the 4th IEEE International Software Metrics Symposium (METRICS), pp. 20–34 (1997)
17. Oracle: Oracle Change Management Pack (2014). http://docs.oracle.com/html/A96679_01/overview.htm
18. IBM: Schema changes (2014). http://pic.dhe.ibm.com/infocenter/db2luw/v10r1/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.dbobj.doc%2Fdoc%2Fc0060234.html
19. IBM: IBM DB2 object comparison tool for Z/OS version 10 release 1 (2012). http://www-01.ibm.com/support/knowledgecenter/SSAUVH_10.1.0/com.ibm.db2tools.gou10.doc.ug/gocugj13.pdf?lang=en
20. Microsoft: SQL management studio for SQL server user's manual (2012). http://www.sqlmanager.net/download/msstudio/doc/msstudio.pdf
21. Snaidero, B.: Capture SQL Server Schema Changes Using the Default Trace. Technical report, MSSQLTips (2015). https://www.mssqltips.com/sqlservertip/4057/capture-sql-server-schema-changes-using-the-default-trace/
22. Microsoft: Microsoft SQL server data tools: Database development zero to sixty (2012). http://channel9.msdn.com/Events/TechEd/Europe/2012/DBI311
23. Terwilliger, J.F., Bernstein, P.A., Unnithan, A.: Worry-free database upgrades: automated model-driven evolution of schemas and complex mappings. In: Elmagarmid, A.K., Agrawal, D. (eds.) Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, 6–10 June 2010, pp. 1191–1194. ACM (2010)
24. Foundation, D.S.: Django (2015). https://www.djangoproject.com/
25. community, S.: South (2015). http://south.readthedocs.org/en/latest/index.html

26. DAINTINESS-Group: Hecate (2015). https://github.com/DAINTINESS-Group/Hecate
27. DAINTINESS-Group: Hecataeus (2015). http://cs.uoi.gr/vassil/projects/hecataeus/index.html
28. Maule, A., Emmerich, W., Rosenblum, D.S.: Impact analysis of database schema changes. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) ICSE, pp. 451–460. ACM (2008)
29. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: Policy-regulated management of ETL evolution. J. Data Semant. **13**, 147–177 (2009)
30. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Aggistalis, K., Pechlivani, F., Vassiliou, Y.: Language extensions for the automation of database schema evolution. In: Cordeiro, J., Filipe, J. (eds.) ICEIS (1), pp. 74–81 (2008)
31. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: Design metrics for data warehouse evolution. In: Li, Q., Spaccapietra, S., Yu, E., Olivé, A. (eds.) ER 2008. LNCS, vol. 5231, pp. 440–454. Springer, Heidelberg (2008)
32. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: HECATAEUS: regulating schema evolution. In: Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE), pp. 1181–1184 (2010)
33. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: Metrics for the prediction of evolution impact in ETL ecosystems: a case study. J. Data Semant. **1**(2), 75–97 (2012)
34. Manousis, P., Vassiliadis, P., Papastefanatos, G.: Automating the adaptation of evolving data-intensive ecosystems. In: Ng, W., Storey, V.C., Trujillo, J.C. (eds.) ER 2013. LNCS, vol. 8217, pp. 182–196. Springer, Heidelberg (2013)
35. Manousis, P., Vassiliadis, P., Papastefanatos, G.: Impact analysis and policy-conforming rewriting of evolving data-intensive ecosystems. J. Data Semant. **4**(4), 231–267 (2015)
36. 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)
37. Mohania, M.: Avoiding re-computation: view adaptation in data warehouses. In: Proceedings of 8th International Database Workshop, Hong Kong, pp. 151–165 (1997)
38. Gupta, A., Mumick, I.S., Rao, J., Ross, K.A.: Adapting materialized views after redefinitions: techniques and a performance study. Inf. Syst. **26**(5), 323–362 (2001)
39. 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. 357–373. Springer, Heidelberg (1998)
40. Golfarelli, M., Rizzi, S.: A survey on temporal data warehousing. IJDWM **5**(1), 1–17 (2009)
41. Wrembel, R.: A survey of managing the evolution of data warehouses. IJDWM **5**(2), 24–56 (2009)
42. Bellahsène, Z.: View adaptation in data warehousing systems. In: Quirchmayr, G., Bench-Capon, T.J.M., Schweighofer, E. (eds.) DEXA 1998. LNCS, vol. 1460, pp. 300–309. Springer, Heidelberg (1998)
43. Bellahsene, Z.: Schema evolution in data warehouses. Knowl. Inf. Syst. **4**(3), 283–304 (2002)
44. Quix, C.: Repository support for data warehouse evolution. In: Gatziu, S., Jeusfeld, M.A., Staudt, M., Vassiliou, Y. (eds.) DMDW. CEUR Workshop Proceedings, vol. 19. CEUR-WS.org (1999)

45. Blaschka, M., Sapia, C., Höfling, G.: On schema evolution in multidimensional databases. In: Mohania, M., Tjoa, A.M. (eds.) DaWaK 1999. LNCS, vol. 1676, pp. 153–164. Springer, Heidelberg (1999)
46. Hurtado, C.A., Mendelzon, A.O., Vaisman, A.A.: Maintaining data cubes under dimension updates. In: Kitsuregawa, M., Papazoglou, M.P., Pu, C. (eds.) ICDE, pp. 346–355. IEEE Computer Society (1999)
47. Hurtado, C.A., Mendelzon, A.O., Vaisman, A.A.: Updating OLAP dimensions. In: Song, I.Y., Teorey, T.J. (eds.) DOLAP, pp. 60–66. ACM (1999)
48. Kaas, C., Pedersen, T.B., Rasmussen, B.: Schema evolution for stars and snowflakes. In: ICEIS (1), pp. 425–433(2004)
49. Wrembel, R., Bebel, B.: Metadata management in a multiversion data warehouse. J. Data Semant. **8**, 118–157 (2007)
50. Eder, J., Koncilia, C., Mitsche, D.: Automatic detection of structural changes in data warehouses. In: Kambayashi, Y., Mohania, M., Wöß, W. (eds.) DaWaK 2003. LNCS, vol. 2737, pp. 119–128. Springer, Heidelberg (2003)
51. Eder, J., Koncilia, C., Mitsche, D.: Analysing slices of data warehouses to detect structural modifications. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 492–505. Springer, Heidelberg (2004)
52. Eder, J., Koncilia, C.: Changes of dimension data in temporal data warehouses. In: Kambayashi, Y., Winiwarter, W., Arikawa, M. (eds.) DaWaK 2001. LNCS, vol. 2114, pp. 284–293. Springer, Heidelberg (2001)
53. 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)