D6.1 Cross-Database Data Migration Techniques Analysis

Cleve, Anthony; Gobert, Maxime

Publication date:
2018

Project Number 780251

D6.1 Cross-Database Data Migration Techniques Analysis Report

Version 1.0
27 June 2018
Final

Public Distribution

University of Namur

Project Partners: **Alpha Bank**, **ATB**, **Centrum Wiskunde & Informatica**, **CLMS**, **Edge Hill University**, **GMV**, **Nea Odos**, **The Open Group**, **University of L′Aquila**, **University of Namur**, **University of York**, **Volkswagen**

Project Partner Contact Information

Alpha Bank Vasilis Kapordelis 40 Stadiou Street 102 52 Athens Greece Tel: +30 210 517 5974 E-mail: vasileios.kapordelis@alpha.gr	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de
Centrum Wiskunde & Informatica Tijs van der Storm Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 9333 E-mail: storm@cwi.nl	**CLMS** Antonis Mygiakis Mavrommataion 39 104 34 Athens Greece Tel: +30 210 619 9058 E-mail: a.mygiakis@clmsuk.com
Edge Hill University Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk	**GMV Aerospace and Defence** Almudena Sánchez González Calle Isaac Newton 11 28760 Tres Cantos Spain Tel: +34 91 807 2100 E-mail: asanchez@gmv.com
Nea Odos Konstantinos Papathanasiou Themistocleous 87 106 83 Athens Greece Tel: +30 210 344 7300 E-mail: kpapathanasiou@neaodos.gr	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L′Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**University of Namur** Anthony Cleve Rue de Bruxelles 61 5000 Namur Belgium Tel: +32 8 172 4963 E-mail: anthony.cleve@unamur.be
University of York Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk	**Volkswagen** Behrang Monajemi Berliner Ring 2 38440 Wolfsburg Germany Tel: +49 5361 9-994313 E-mail: behrang.monajemi@volkswagen.de

Document Control

Version	Status	Date
0.1	Document outline	1 Feb 2018
0.2	Bibliography collection	5 March 2018
0.3	Incomplete draft	30 April 2018
0.4	Full draft for internal review	10 June 2018
1.0	QA review	27 June 2018

Table of Contents

Executive Summary

Today, an increasing number of organizations are considering NoSQL database engines as a migration target for existing (relational-based) software systems or as a platform for planned future systems. Success stories around the use of NoSQL systems have spurred widespread interest in these new technologies and even raised the question as to whether there is a continued future role for relational databases. At the same time, the hype surrounding NoSQL databases has created a significant amount of confusion about trade-offs between relational and NoSQL-based solutions, and some projects that started with NoSQL found the need to migrate parts of their system in the other direction, namely to a relational foundation.

The consensus that seems to be emerging from the relational/NoSQL debate indicates that the two types of systems address substantially different classes of problems and that they should be selected accordingly. At the same time, today's information-intensive businesses often exhibit evolving data management requirements that cross-cut the problem domains of relational and NoSQL systems. There is a lack of methodological guidance and tool support for *migrating* data between relational and NoSQL systems and for effectively *bridging* both types of systems in the context of data-intensive system evolution.

Migrating data from a database platform to another one should, in an ideal world, only impact the database component of the software system. Unfortunately, the database most often has a deep influence on other components, such as the application programs. Two reasons can be identified. First, the programs invoke data management services through an API that typically relies on complex and highly specific protocols. Changing the database platform, and therefore its protocols, requires to rewrite the database manipulation code sections. Second, the database schema is the technical translation of its conceptual schema through a set of rules that is dependent on the underlying data model of the database management system. Porting the database to another database management system, and therefore to another data model, generally requires another set of rules, that produces a significantly different database schema. Consequently, the code of the programs often has to be adapted to this new schema. Clearly, this data migration process leads to non trivial database (schemas and data) changes, as well as programs adaptations.

The TYPHON project, via its Work Package 6, has the ambition to develop a methodology and technical infrastructure to support the graceful evolution of hybrid polystores, where multiple, possibly overlapping relational and NoSQL databases may co-evolve in a consistent manner. This includes, among others, the development of methods and tools for (1) migrating data across paradigms and across platforms and (2) evolving the data organisation and distribution in hybrid persistence architectures. The proposed approaches and supporting tools aim to be as transparent as possible for the client application programs manipulating the evolving hybrid polystores.

This document aims at identifying and discussing existing approaches, techniques and tools that could be inspiring, reused or extended in order to address these challenges.

1 Introduction

Today, an increasing number of organizations are considering NoSQL database engines as a migration target for existing (relational-based) software systems or as a platform for planned future systems. Success stories around the use of NoSQL systems have spurred widespread interest in these new technologies and even raised the question as to whether there is a continued future role for relational databases. At the same time, the hype surrounding NoSQL databases has created a significant amount of confusion about trade-offs between relational and NoSQL-based solutions, and some projects that started with NoSQL found the need to migrate parts of their system in the other direction, namely to a relational foundation [24].

The consensus that seems to be emerging from the relational/NoSQL debate indicates that the two types of systems address substantially different classes of problems and that they should be selected accordingly. At the same time, today's information-intensive businesses often exhibit evolving data management requirements that cross-cut the problem domains of relational and NoSQL systems. There is a lack of methodological guidance and tool support for *migrating* data between relational and NoSQL systems and for effectively *bridging* both types of systems in the context of data-intensive system evolution.

Migrating data from a database platform to another one should, in an ideal world, only impact the database component of the software system. Unfortunately, the database most often has a deep influence on other components, such as the application programs. Two reasons can be identified. First, the programs invoke data management services through an API that typically relies on complex and highly specific protocols. Changing the database platform, and therefore its protocols, requires to rewrite the database manipulation code sections. Second, the database schema is the technical translation of its conceptual schema through a set of rules that is dependent on the underlying data model of the database management system. Porting the database to another database management system, and therefore to another data model, generally requires another set of rules, that produces a significantly different database schema. Consequently, the code of the programs often has to be adapted to this new schema. Clearly, this data migration process leads to non trivial database (schemas and data) changes, as well as programs adaptations.

The TYPHON project, via its Work Package 6, has the ambition to develop a methodology and technical infrastructure to support the graceful evolution of hybrid polystores, where multiple, possibly overlapping relational and NoSQL databases may co-evolve in a consistent manner. This includes, among others, the development of methods and tools for (1) migrating data across paradigms and across platforms and (2) evolving the data organisation and distribution in hybrid persistence architectures. The proposed approaches and supporting tools aim to be as transparent as possible for the client application programs manipulating the evolving hybrid polystores.

This document aims at identifying and discussing existing approaches, techniques and tools that could be inspiring, reused or extended in order to address these challenges.

1.1 Purpose of the deliverable

This document presents the work that has been done with respect to the following task of Work Package 6 as presented in the TYPHON Description of Work:

Task 6.1: This task will identify, analyze and compare existing techniques and tools for cross-database data migration. Although such an analysis has already been performed at the proposal preparation stage, an additional survey will be necessary at the start of the project to ensure the timeliness of subsequent work in this work package.

As described in the Description of Work, this deliverable *will consist of a detailed report about the existing techniques and tools for cross-database data migration.*

1.2 Structure of the deliverable

The remainder of this deliverable is structured as follows:

1. Section 2 introduces the main concepts used in this document and defines the main artifacts and processes involved in database evolution in general and database migration in particular;
2. Section 3 elaborates on general methodologies and approaches to database evolution/migration, considering the *entire* evolution process;
3. Section 4 describes possible approaches to database reverse engineering, which often constitutes the initial phase of the database evolution/migration process;
4. Section 5 elaborates on existing approaches and techniques related to *intra-paradigm* database evolution scenarios, i.e., without database platform change;
5. Section 6 presents existing approaches and techniques related to *inter-paradigm* database evolution scenarios, i.e., involving a database platform change;
6. Section 7 focuses on approaches and techniques supporting the sub-process of adapting application programs and queries to an evolving database;
7. Section 8 summarizes and classifies the surveyed approaches by means of a comparative table;
8. Section 9 gives concluding remarks.

1.3 Relationship to other TYPHON deliverables

The approaches, techniques and tools described in the present deliverable will be inspiring when designing the Hybrid Polystore Schema Evolution Methodology and Tools (deliverable D6.2), the Hybrid Polystore Data Migration Tools (deliverable D6.3) and the Hybrid Polystore Query Evolution Tools (deliverable D6.4). This document also relates to the (upcoming) deliverables of other TYPHON work packages, in particular those related to the TyphonML modeling language (WP2), the TyphonQL query language (WP4) and the hybrid polystore monitoring tools (WP5). The deliverables from WP2 will trigger/instruct the needed data migrations with respect to schema evolutions defined at TyphonML level. The deliverables from WP4 will constitute the basis to develop the data and query migration tools. The deliverables from WP5 will be used to automatically recommend polystore reconfigurations by enabling data access analysis.

1.4 Contributors

The main contributor of this deliverable is University of Namur. All project partners contributed to this deliverable, by collaboratively eliciting the use case and technical requirements for Work Package 6, as well as by providing feedback on earlier versions of this document.

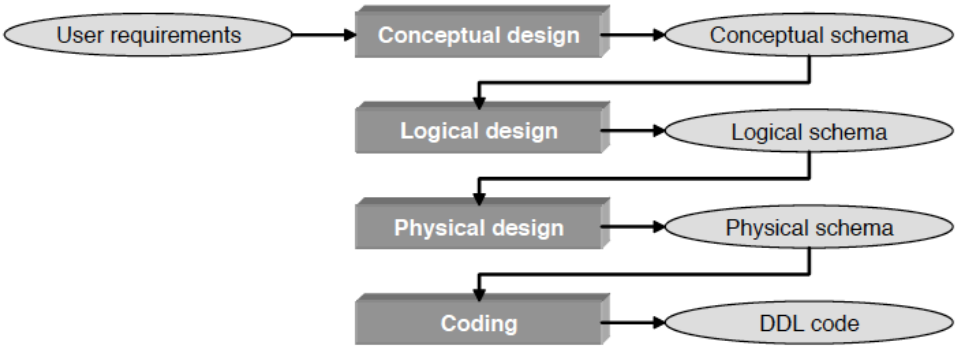

Figure 1: Standard database design processes.

2 Preliminaries

This section aims at providing the reader with an introduction to the main concepts used in this document.

2.1 Database design

The process of designing and implementing a database that has to meet specific user requirements has been described extensively in the literature [8] and has been available for several decades in standard methodologies and CASE tools. As shown in Figure 1, database design is typically made up of four main subprocesses:

(1) **Conceptual design** is intended to translate user requirements into a *conceptual schema* that identifies and describes the domain entities, their properties and their associations in a platform-independent way. This abstract specification of the database collects all the information structures and constraints of interest.
(2) **Logical design** produces an operational *logical schema* that translates the constructs of the conceptual schema according to a specific technology family, in principle, without loss of semantics.
(3) **Physical design** augments the logical schema with performance-oriented constructs and parameters, such as indexes, buffer management strategies or lock management policies. The output of this process is the *physical schema* of the database.
(4) **Coding** translates the physical schema (and some other artefacts) into the DDL (*Data Definition Language*) code, compliant with the target database management system. Structural DDL declaration code as well as components such as checks, triggers and stored procedures are written/generated to implement the information structures and constraints of the physical schema.

Database schemas express the structures and constraints of a database. As mentioned above, they are usually classified according to the level of abstraction they belong to: *conceptual*, *logical* and *physical*.

- **Conceptual schemas** A conceptual schema is a *platform-independent* model of the database. It mainly specifies *entity types*, *relationship types* and *attributes*. Entity types represent the main concepts of the application domain. They can be organized into *is-a* hierarchies, organizing supertypes and subtypes. Relationship types represent relationships between entity types. Attributes represents common properties of the entity type instances.

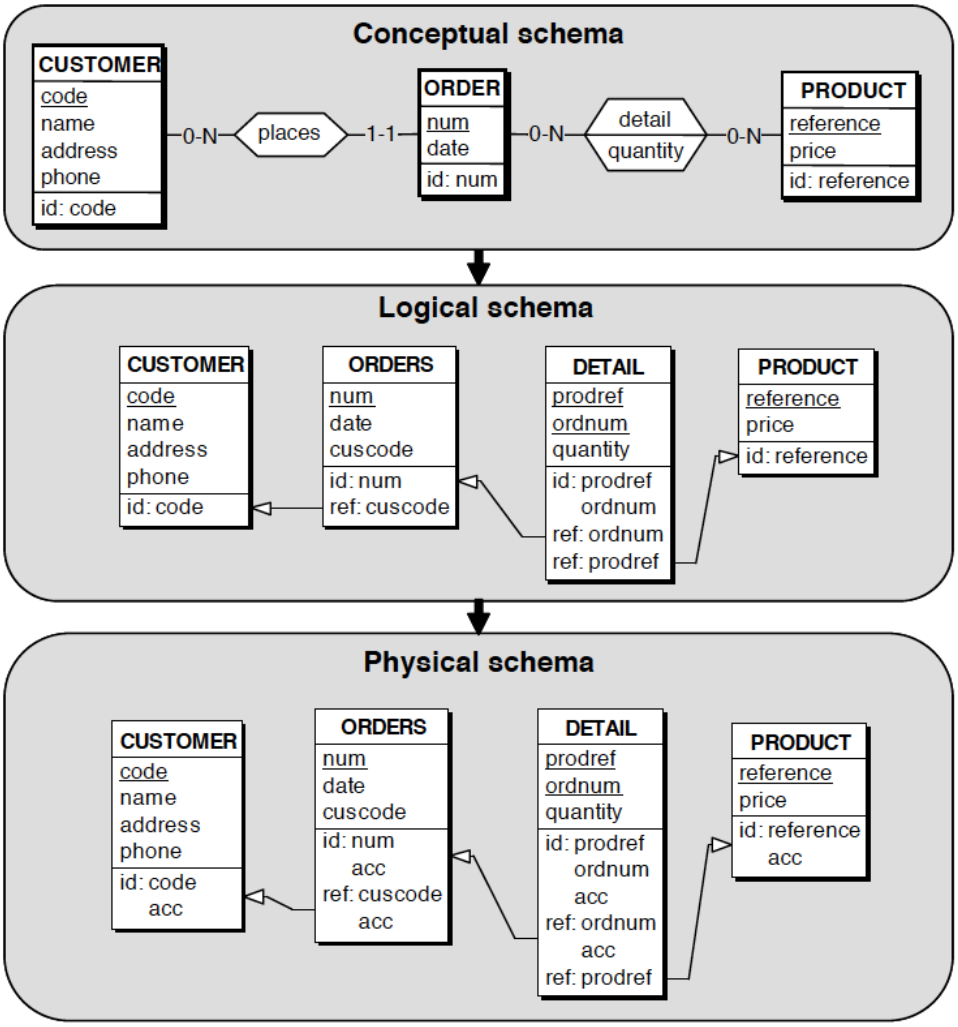

Figure 2: Example of conceptual, logical and physical schemas of a relational database.

- **Logical schemas** A logical schema is a *paradigm-dependent* data structure definition, that must comply with a given data model. The most commonly used families of models include the relational model, the network model (CODASYL), the hierarchical model (IMS), the standard file model (COBOL, RPG, BASIC, etc.), the shallow model (TOTAL, IMAGE), the XML model, the object-oriented model, the object-relational model and the NoSQL models (column-oriented, graph-oriented, document-oriented, etc.).
- **Physical schemas** A physical schema is a logical schema enriched with all the information needed to implement *efficiently* the database on top of a given data management system. This includes *platform-dependent* technical specifications such as indexes, physical device and site assignment, page size, file size, buffer management or access right policies. Due to their large variety, it is not easy to propose a general model covering all possible physical constructs.

Figure 2 gives an example of conceptual, logical and physical schemas, expressed in the Generic Entity-Relationship (GER) data model [41]. In this example, all the constructs belonging to the conceptual schema have been translated into equivalent constructs in the logical schema.

2.2 NoSQL data models

The number of NoSQL systems is still growing and they represent now nearly half (170) of the total number of database management systems on the market [1]. But, in contrast with relational database management systems, NoSQL engines do not rely on the same theoretical data model.

Those systems can be grouped into four different data models, each having its specific requirements and advantages. We further explain and summarize them below based on a survey provided by Hecht and Jablonski [45].

2.2.1 Key Value Stores

The Key value store model is the most "schema-less" model among NoSQL systems. It is indeed based on a simple key-value pair, with no constructs allowing to define explicit relationships between data instances. The values are stored in byte arrays and therefore the only way to retrieve data is by means of the keys. Table 1 shows an example of such a representation. They are useful for very simple operations and basic application usage as they only provide *put, get* and *delete* operations. Other operations or queries have to be managed in the application code. Example platforms based on the key value data model are Redis[83], ArangoDB[4] and Riak KV[84].

Key	**Value**
ABDD	11101010
DBFG	10101010
FHJD	11100101

Table 1: Example of key value data model instance.

2.2.2 Document stores

The document store data model has a similar structure as the key value model. Data is stored as key-value pairs but they are wrapped in a JSON like document. This model offers the flexibility of a schema less data store but it also helps the developer to query data, unlike key value data store the values can be queried in a more complex way and provide developers the means to express user friendly queries. But unlike SQL for relational model, the NoSQL ecosystem does not provide a standard query language, each system has its own, forcing the developer to learn a new one for each system he uses.

Document store model have a set of mapping rules to map a relational model. MongoDB for example provides a set of example on their website [2]. Table 2 lists the mapped concepts.

The flexibility of the document model allows to have multiple different document in a same collection. Figure 3 shows three documents, each of them have a different set of attributes but are in the same collection.

MongoDB[71], Couchbase[25], CouchDB[26] and ArangoDB[4] are examples of document store model database system.

[1] https://db-engines.com/en/ranking_categories
[2] https://docs.mongodb.com/manual/reference/sql-comparison/

SQL Concepts	MongoDB Concept
database	database
table	collection
row	document
index	index
table joins	$lookup function, embedded documents design
primary key	_id field
group by	aggregation pipeline

Table 2: Table of equivalent SQL - MongoDB concepts

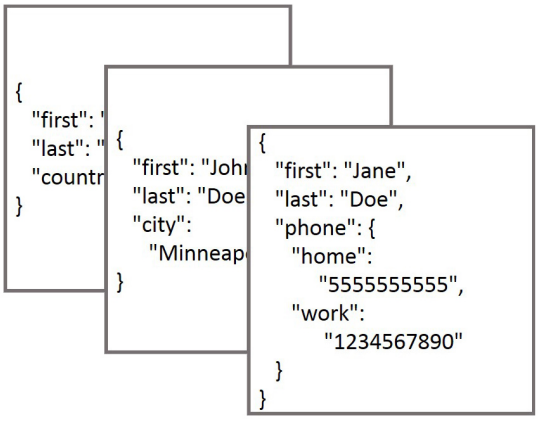

Figure 3: Document Oriented data model.

2.2.3 Column Family stores

The column family store is organized as a map of key value pairs. This way one can group key values (or columns) together and creating a family (or cluster), as shown in figure 4. Similarly to key value store, the values cannot be queried. Technically the data is stored ordered by keys and retrieving of big amount of data is more efficient, this is also used for partitioning data on distributed servers. Systems implementing this data model are HBase[44] or Cassandra[13].

2.2.4 Graph databases

The last important data model category is the graph database. This has been built in order to specifically manage heavily linked data. It can consist of simple triple values like RDF or more complex structures containing key value pairs. Graph databases can be queried in two different ways, either by trying to find a part of the graph that matches a criteria or by exploring the graph by starting in a specific node. It is also possible to express constraints that limit the type of edges applicable to certain type of nodes. An advantage that no other data models gives is that it can also be queried with a standard language (SPARQL) to more than one graph database management system. Neo4j[73], GraphDB[38] and ArangoDB[4] can be used for such implementation.

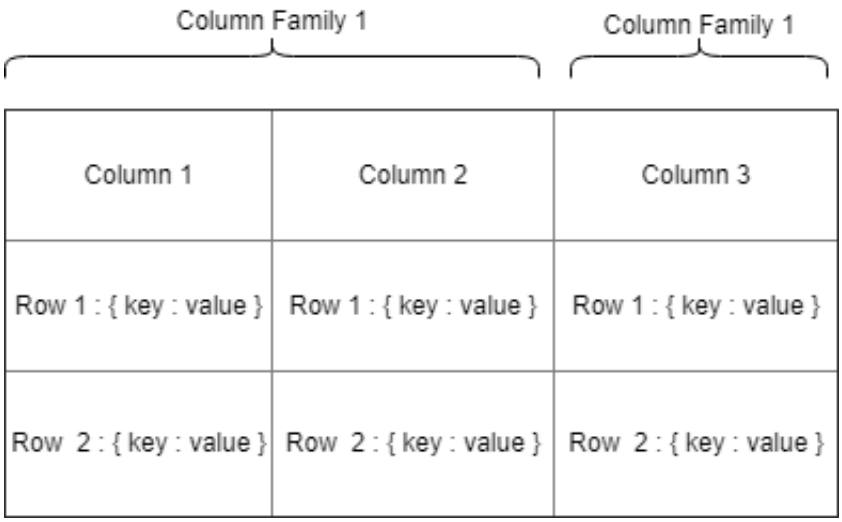

Figure 4: Column Oriented data model.

2.3 NoSQL database design

NoSQL databases are often based on a schema-less data models and are therefore oriented towards developers. The underlying goal is to develop applications faster, while taking less care of the underlying database structures, which otherwise are more rigid and harder to evolve in combination with the programs. However, those advantages can eventually lead to poor performance as well, since a class embedding, redundancy or bad design decisions may significantly affect scalability, performance and consistency.

Atzeni [6, 5] introduce NoAM (NoSQL Abstract Model), an abstract data model for NoSQL databases, which exploits the commonalities of various NoSQL data models. They propose a database design methodology for NoSQL systems [12] based on NoAM, which aims to be (partially) independent of the specific target NoSQL platform. NoAM is used to specify a system-independent representation of the application data. This intermediate, pivot representation can then be implemented in specific NoSQL database platform. The proposed method consists of four main steps:

- **Aggregate design**. Following use cases different entities are grouped together. This is consistent with the Domain Driven Design methodology.
- **Aggregate partitioning**. Performance requirements guide the smaller partition of aggregates.
- **High level NoSQL database design**. Based on conceptual modelling. It provides concepts such as Entry per Aggregate Object (EAO) or Entry per Top-level Field (ETF)
- **Implementation**. Mapping intermediate structures to the target datastore.

Another approach to designing and implementing NoSQL database is proposed by Abdelhedi *et al.* [1], they use a model driven approach and transformation rules on a conceptual database schema in order to create a NoSQL (specifically a column oriented) logical schema and then a physical NoSQL schema. It is a two step process, a first one is to create a model to conceptualize the data independently of all technical and data model aspects (PIM *Platform Independent Model*). After that is the PSM *Platform Specific Model* which represents the data in regards of a specific data model, in this paper it is the column-oriented model. The transition between the two model is done via a set of transformation rules in QVT language (Query/View/Transformation) following the OMG specification.

2.4 Database evolution scenarios

Cleve [17] proposes to classify typical database evolutions according to the following three dimensions:

- **Structural dimension**, that concerns the modification of the database structures (schemas);
- **Semantic dimension**, which relates to the evolution of the semantics of the schemas;
- **Platform/Language dimension**, addressing the replacement of the data description and manipulation languages.

As an example, Figure 5 instantiates those classification dimensions in the case of three popular evolution scenarios: database *migration*, database *refactoring* and database *integration*.

	Structural	**Semantic**	**Platform**
Database migration	✓		✓
Database restructuring	✓	(✓)	
Database integration	✓	✓	(✓)

Figure 5: Database evolution scenarios classified according to three dimensions.

- **Database migration** consists of the substitution of a data management technology for another one. This scenario raises two major issues. The first one is the conversion of the database schema and instances to a new data management system. The database structure is often modified, but both source and target schema should cover the same *universe of discourse* (i.e., structural modifications but no semantic change). The second problem concerns the adaptation of the application programs to the migrated database schema and to the target data management system.
- **Database restructuring** does not involve any language replacement, but only structural modifications. Depending on the type of schema transformations applied, the target schema may convey another semantics. For instance, renaming a SQL column does not induce any semantic change while adding a new column does.
- **Database integration** aims at obtaining a single database from heterogeneous databases that belong to the same application domain. The resulting database structure and semantics typically differ from the ones of the input databases. The databases to be integrated are not always of the same platform.

Below, we further present and analyze each of the three identified dimensions of database evolution.

2.4.1 Structural dimension

The structural dimension is concerned with the evolution of the database structures. This regroups modifications applied to the conceptual, logical and physical schemas. As proposed by Hick and Hainaut [47], we can classify database schema evolutions scenarios according to the schema *initially* modified:

- **Conceptual modifications** typically translate changes in the functional requirements of the information system into conceptual schema changes.
- **Logical modifications** do not modify the requirements but adapt their platform-dependent implementation in the logical schema.
- **Physical modifications** aim at adapting the physical schema to new or evolving technical requirements, like data access performance.

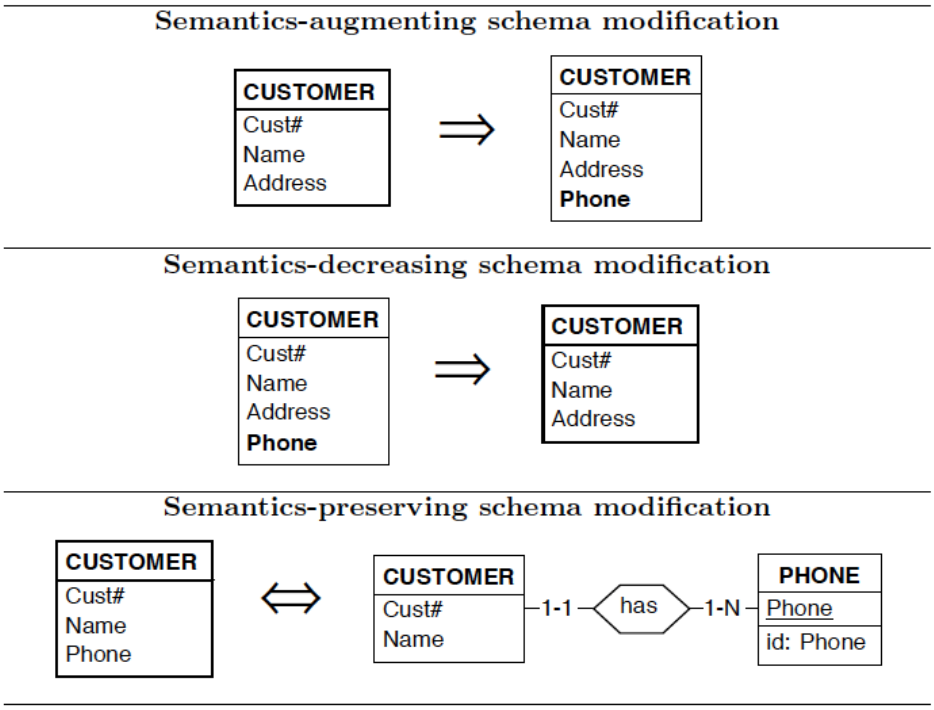

Figure 6: Examples of semantics-augmenting, semantics-decreasing and semantics-preserving schema modifications.

2.4.2 Semantic dimension

The semantic dimension captures the impact of a given database evolution scenario on the informational content of the target database. In other words, it aims at indicating whether the evolution involves:

- *Semantics-augmenting* schema modifications (S^+).
- *Semantics-decreasing* schema modifications (S^-);
- *Semantics-preserving* schema modifications ($S^=$);

A semantics-preserving schema modification ($\in S^=$) is commonly called *schema refactoring* [3]. The modifications belonging to the two other categories (S^- and S^+) can be regrouped under the term *schema semantic adaptation*. Examples of the three kinds of schema modifications are given in Figure 6.

2.4.3 Platform/Language dimension

The platform or language dimension intends to characterize a database evolution scenario in terms of platform/language change. The following possible cases can be distinguished:

- **Intra-platform database evolution**: the database evolution does not involve the replacement of the data management system.
- **Inter-platform database evolution**: the database evolution requires the replacement of the data management system with another one. Two sub-cases can occur:

- **Intra-paradigm platform change**: the source and target database platforms belong to the same paradigm. A typical example is the migration of a relational database from MySQL to PostgreSQL.
- **Inter-paradigm platform change**: the database evolution relies on a database paradigm switch, i.e., the source and target database platforms belong to different paradigms. This is the case, for instance, when migrating a relational database (e.g. MySQL) to a NoSQL platform (e.g. MongoDB).

2.5 Database evolution processes

Database evolutions are typically composed of the following chain of sub-processes.

1. **Database reverse engineering** is the usual initial step of database evolution, aiming at *understanding* the source database subject to evolution. This process is required in the (very) frequent situation in which the source database is not (well-)documented. In Section 4 we focus on this initial step and present the main existing approaches to database reverse engineering.
2. **Impact analysis** aims to evaluate the impact (chain) of the desired schema modification(s) on the related artefacts (other schemas, data instances and programs).
3. **Database evolution** is the evolution of the database component, itself comprising two subprocesses:
   (a) **Database schema change** consists of applying the necessary change(s) to a database schema at a given level of abstraction and to adapt the schemas belonging to the other abstraction levels[3].
   (b) **Data adaptation** concerns the adaptation of the data instances to the modified schemas.
4. **Program adaptation** aims to adapt the application programs to the target database schema and platform. It involves, in particular, the adaptation of the *queries* that are used by the programs to manipulated the database subject to changes. In Section 7 we present existing works analyzing and supporting the co-evolution of databases and application programs.

Note that depending on the nature of the particular database evolution and depending on the methodology chosen to achieve this evolution, some of the above processes may be useless or skipped on purpose.

[3]For instance, the modification of the conceptual schema typically necessitates the adaptation of the logical schema, physical schemas and DDL code.

3 General database migration methodologies

Migrating large database-centric software systems towards a new database platform has long been considered as one of the most complex and failure-prone processes. Database platform migration consists of deriving a target database from a source database and in further adapting the software components accordingly [11].

3.1 Existing approaches

Several system migration strategies have been identified and described in the research literature, notably by Brodie and Stonebraker [11]. They can be classified according to several dimensions. Identifying two major components, namely the data and the programs, we can distinguish two families of strategies, according to which component is migrated first.

- **Database first strategies**. First, the source database is migrated, so that new programs can be developed on the target platform. Later on, the source programs are migrated in order to manipulate the new database. In the mean time, they either keep using the source database, which is synchronized with the target database, or they access the latter through some sort of *wrappers*.
- **Database last strategies**. First, the programs are migrated to the new database platform. They then use the source database through wrappers. New applications access the source database through the same interface. When all the applications have been converted, the database itself is migrated.

The second classification dimension concerns the time frame within which the database platform replacement is carried out. One typically identifies two main families.

- **Big bang approach**. The target system, comprising the data and the programs, replaces the source system in *one* step. Most generally, the substitution is carried out in a very short time, typically a few days, so that both systems run with no overlap.
- **Chicken little approach**. The database and the applications are migrated incrementally, to mitigate the risks and allow continuous regression testing of each migrated system subset.

Hainaut *et al.* [42] studies the migration of a (legacy) system towards a more modern data management technology. They develop a two-dimensional reference framework that identifies six representative migration strategies, which are further analyzed to derive methodological requirements. They show that transformational techniques are particularly suited to drive the whole database migration process. They also study the problem of program conversion. Some program conversion strategies appear to minimize the program adaptation effort, and therefore are sound candidates to develop practical methodologies.

Considering that a database is made up of two main components, namely its schema(s) and its contents (the *data*), the data migration process usually comprises three main steps: (1) *schema conversion*, (2) *data conversion* and (3) *program conversion*. Figure 7 depicts the usual organization of the *database-first* migration process, that is made up of subprocesses that implement those three steps.

The schema conversion process usually produces a formal description of the mapping between the objects of the source schema (S) and those of the target schema (S'). This mapping is then used to convert the data and the programs. Practical methodologies differ in the extent to which these processes are (partly) automated.

- **Schema conversion** is the translation of the source database structure, or schema, into an equivalent database structure expressed in the target technology. Both schemas must convey the same semantics, i.e., all the source data should be stored into the target database without loss.

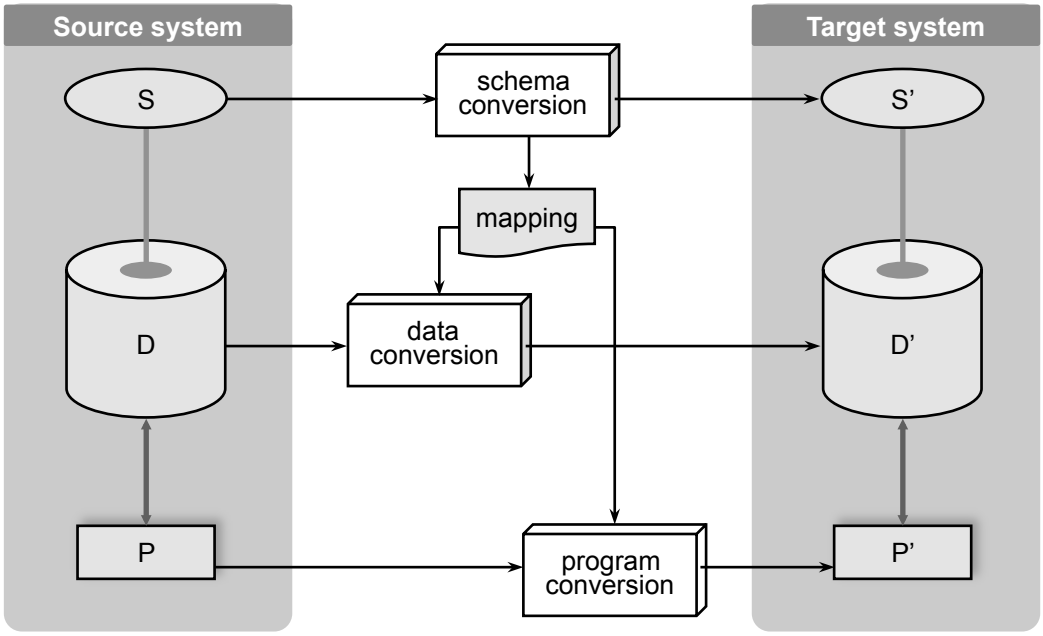

Figure 7: Overall view of the *database-first* database migration process (adapted from [42]).

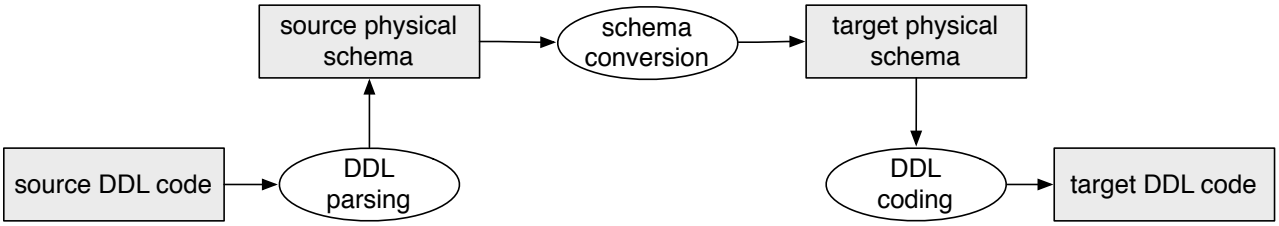

Figure 8: Physical, one-to-one schema conversion strategy.

- **Data conversion** is the migration of the data instances from the source database to the target one. This migration involves data transformations that derive from the schema transformations described above.
- **Program conversion**, in the context of database migration, is the modification of the program so that it now accesses the migrated database instead of the source data. The functionalities of the program are left unchanged, as well as its programming language and its user interface.

Hainaut *et al.* [42] identify two extreme database conversion strategies leading to different levels of quality of the transformed database. The first strategy, called *physical conversion*, consists in translating each construct of the source database schema into the *closest* constructs of the target database model, without attempting any semantic interpretation. The process, depicted in Figure 8, remains quite cheap, but it usually leads to poor quality databases with no added value.

The second strategy, called *conceptual conversion*, aims at first recovering the precise semantic description (i.e., its conceptual schema) of the source database, through reverse engineering techniques, then in designing the target database from this conceptual schema through a standard database design methodology. The target database is of high quality according to the expressiveness of the target platform model and is also fully

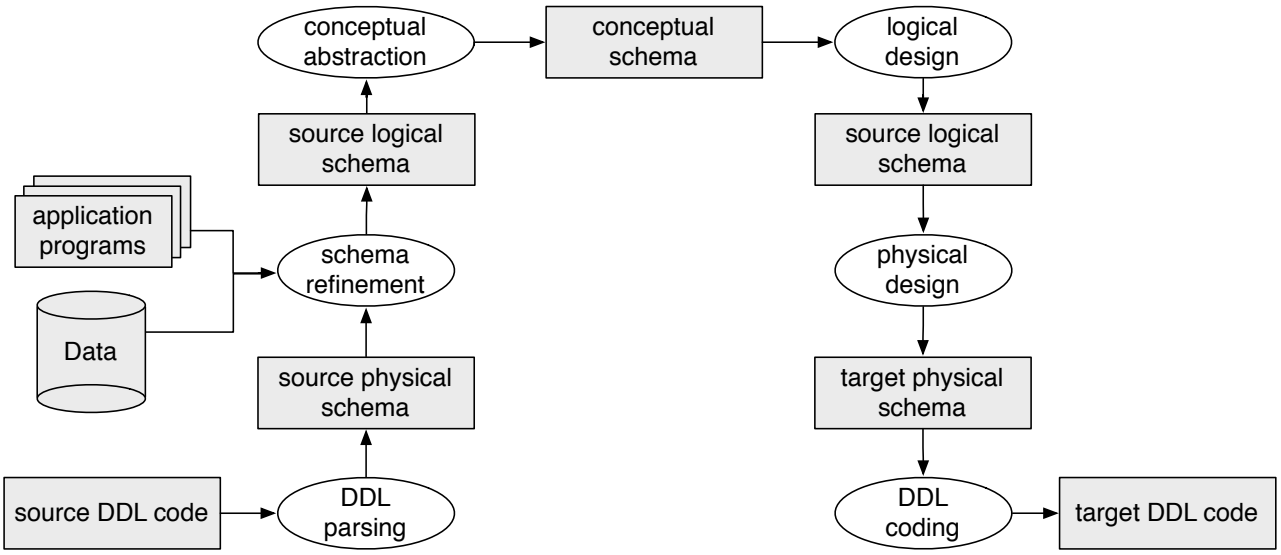

Figure 9: Conceptual schema conversion strategy.

re-documented. However, the conceptual conversion approach is usually more expensive than the physical conversion approach, due to the database reverse engineering step.

Other general methodologies have been proposed to migrate (legacy) systems towards a more modern database platform. The Varlet project [49] adopts a typical two phase migration process comprising a reverse engineering process phase followed by a standard database implementation. The approach of Jeusfeld [50] is divided into three parts: mapping of the original schema into a meta model, rearrangement of the intermediate representation and production of the target schema. Bianchi *et al.* [10] propose an iterative approach to legacy database migration. This approach aims at eliminating the ageing symptoms of the legacy database [101] when incrementally migrating the latter towards a modern platform.

Several authors have addressed database migration between particular source and target database paradigms. Among them, Menhoudj and Ou-Halima [66] present a method to migrate the data of COBOL legacy systems into a relational database management system. A hierarchical-to-relational database migration approach is proposed by Meier *et al.* [65, 64]. General approaches to migrate relational database to object-oriented technology were proposed by Behm *et al.* [9] and by Missaoui *et al.* [70].

3.2 General evolution methodology in TYPHON

The evolution of Typhon polystores will follow a *database first* evolution strategy, according to which the users will first apply a schema change at the abstraction level of TyphonML. The related physical schemas, data instances and TyphonQL queries will then be adapted by the TYPHON migration tools. In the case of *semantics-preserving* schema changes, such adaptations can be automatically supported. In the case of *non-semantics-preserving* schema changes, automated adaptation of data and queries is, by definition, not possible. Only the impact analysis process can be tool-supported, via the automated identification of invalid data and invalid TyphonQL queries that will then need to be *manually* adapted (when possible) or simply deleted from the system.

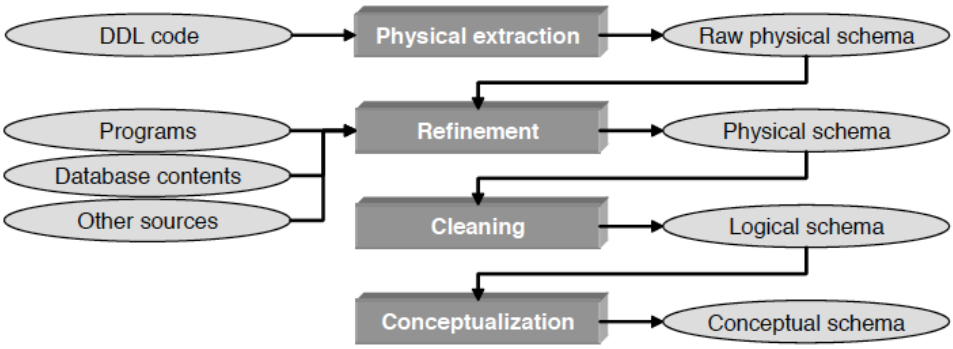

Figure 10: Standard database reverse engineering processes.

4 Database reverse engineering

Database platform migration often relies on the availability of up-to-date documentation of the current version of the database. However, in practice, the documentation may be incomplete, obsolete or simply missing. As illustration, the conceptual, logical and physical database schemas are often needed to ensure the evolution task. However, the DDL code often constitutes the only available documentation of the data structures and constraints. Furthermore, this DDL code may be incomplete since some data structures and constraints are not explicitly declared in the DDL code. This may be the case, for instance, for implicit foreign key constraints. Therefore, the process of recovering those implicit properties may prove indispensable but this requires an additional, possibly significant effort.

Chikofsky [16] defines data reverse engineering as "*a collection of methods and tools to help an organization determine the structure, function, and meaning of its data*". Database Reverse Engineering (DBRE) is *the process through which the logical and conceptual schemas of a legacy database, or of a set of files, are reconstructed from various information sources such as DDL code, data dictionary contents, database contents or the source code of applications that use the database.* [43].

As depicted in Figure 10, database reverse engineering typically comprises the following four sub-processes:

(1) **Physical extraction** consists in parsing the DDL code in order to extract the raw physical schema of the database.
(2) **Refinement** enriches the raw physical schema with additional structures and constraints elicited through the analysis of the application programs and other sources of information.
(3) **Cleaning** removes the physical constructs (such as indexes) for producing the logical schema.
(4) **Conceptualization** aims at deriving the conceptual schema that the logical schema obtained implements.

4.1 Existing approaches

Several authors have proposed approaches to support database reverse engineering. The proposed techniques exploit different information sources during the refinement process.

- **Database schema analysis** [72, 62, 79]. Analyzing the database schema structures may help identify hidden constructs such as relationships and hierarchies between data.

- **Data analysis** [15, 61, 104, 75]. Mining the database contents can be used in two ways. Firstly, to discover implicit properties, such as functional dependencies and foreign keys. Secondly, to check hypothetic constructs that have been suggested by other means.

- **Graphical user interface analysis** [97, 82]. Forms, reports and dialog boxes are user-oriented views on the database that exhibit spatial structures, meaningful names, explicit usage guidelines and, at runtime, data population and error messages that can provide information on data structures and constraints.

- **Static program analysis** [77, 31, 21]. Analysis, such as data-flow graph exploration, can bring valuable information on field structure and meaningful names. More sophisticated techniques such as program slicing can be used to identify complex constraint checking.

- **Dynamic program analysis** [40, 20, 2, 23]. In the case of highly dynamic program-database interactions, the database queries may only exist at runtime. Hence recent techniques allow to capture and analyze SQL execution traces in order to retrieve structural information.

- **Evolution history analysis** [19]: Schema changes occur often in a database lifetime, those schema changes can have a significant informative value in the context of reverse engineering. For instance, Cleve *et al.* [19] introduced the concept of *global historical schema* which is an aggregated schema of all previous versions of a database schema. They then analyse this historical schema in order to better understand the current version of the database schema.

- **Hybrid methods**: the database reverse engineering process may greatly benefit from the cross-analysis of *several* artifacts, as shown in several papers [22, 69]

4.2 Database reverse engineering in TYPHON

The database reverse engineering step is out of the scope of the TYPHON approach as far as database evolution/migration is concerned. However, before adopting the TYPHON technologies to manipulate hybrid polystores it might be necessary, as an initial step, to understand the structures and constraints of each *pre-existing* database. Hence, we considered the database reverse engineering process in the present survey.

5 Intra-platform database evolution

In a multi platform and multi paradigm such as TYPHON schema evolution can have a very broad definition. In this section we particularly address the schema evolution in an *intra-platform* or *cross-database* point of view. This means that we only consider here the different research done regarding the evolution of database by keeping the same platform and therefore within the same data model, a scenario also called *schema evolution*.

The research literature related to schema evolution is very large. Rahm *et al.* [81] built an online bibliography that consists of a comprehensive and up-to-date collection of publications related to schema evolution. They categorized publications along multiple hierarchical dimensions. The online bibliography is not strictly limited to database schema evolution; related fields such as ontology evolution, software evolution and workflow evolution are also covered.

In this section, we briefly summarize the most relevant schema evolution approaches and methods in the context of the TYPHON project.

5.1 Existing approaches

5.1.1 Analyzing schema evolution

Sjøberg [88] studied the schema evolution history of a large-scale medical application and showed, by using a thesaurus tool, that even a small change to the schema may have major consequences for the rest of the application code. The study reveals that schema changes are significant both in the development period and after the system has become operational. The consequences of the schema changes on the application programs have been measured. In particular, the tool provides information about how many screens, actions, queries, etc. may be affected by a possible schema change. The results confirm that change management tools are needed.

Curino *et al.* [28] present a study of the structural evolution of the Wikipedia database, with the aim to extract both a micro-classification and a macro-classification of schema changes. They also study the frequency distribution of those schema changes. In addition to a schema evolution statistics extractor, the authors propose a tool that operates on the differences between subsequent schema versions and semi-automatically extracts the set of possible schema changes that have been applied. In this study, a period of four years has been considered, corresponding to 171 successive versions of the Wikipedia database schema. Their study shows the need for automated support to schema evolution.

Lin and Neamtiu [58] showed the impact of databases schema changes to application programs. They investigated two case studies showing that evolution of the database schema could cause loss of performance or failure to the program related to those databases. They showed the importance of co-evolving the programs and how it was done in particular case studies. Those program adaptations were carried out manually in a non efficient way.

Vassiliadis *et al.* [99, 100] studied the evolution of individual database tables over time in eight different software systems. They report on their observations on how evolution-related properties, like the possibility of deletion, or the amount of updates that a table undergoes, are related to observable table properties like the number of attributes or the time of birth of a table. Through a large-scale study on the evolution of database, they also tried to determine whether Lehman's laws of software evolution hold for evolving database schemas as well [89]. They conclude that the essence of Lehman's laws remains valid in this context, but that specific mechanics significantly differ when it comes to schema evolution.

Schema changes occur often in a database lifetime, those schema changes can have a significant informative value in the context of reverse engineering. Cleve *et al.* [19] introduce the concept of *global historical schema*

which is an aggregated schema of all previous versions of a database schema. They then analyse this schema in order to better understand the current version of the database schema and, therefore, facilitate future schema changes.

5.1.2 Supporting schema evolution

Hick and Hainaut [46, 47] propose the DB-MAIN approach to database schema evolution. This approach relies on a generic database model, namely the GER model [41], and on transformational paradigm that states that database engineering processes can be modeled by (chains of) schema transformations. Indeed, a transformation provides both structural and instance mappings that formally define how to jointly modify database structures and related data instances. The authors describe both a complete and a simplified, more pragmatic version of their approach, and compare their respective merits and drawbacks.

Table 3, inspired from [46], provides a semantic classification of the schema modification operators considered by the DB-MAIN approach. The presented operators are based on the constructs of the GER model [41].

Database evolutions may involve schema modifications that can in turn impact the data instances and the database queries. Adapting data and queries to evolving schemas may constitute a long, risky and often manual process for database administrators. In [27], Curino *et al.* present PRISM++, a system that supports the database evolution process by evaluating the impact of schema modifications on queries and on data. PRISM++ then help developers with the rewriting of historical queries and the migration of related data, thereby reducing the downtime of the system by reducing manual effort. They achieve this by defining a set of *Schema Modification Operators* (SMOs) representing atomic schema changes, and they link each of these operators with modification functions for data and queries.

The evolution operators considered by PRISM++ are inspired by the operators defined by Ambler and Sadalage [3]. They defined six main categories of operators, namely *transformation*, *structure refactoring*, *referential integrity refactoring*, *architecture refactoring*, *data quality refactoring* and *method refactoring*. In order to be even more precise, it is possible to further classify those operators into finer-grained atomic operators, as done by Curino *et al.*.

In the same spirit, Qiu *et al.* [80] propose an exhaustive list of 24 schema change operators, each corresponding to an atomic DDL query. Those are detailed in Figure 11, taken from [80].

More recent approaches and studies have focused on the evolution of NoSQL databases. Scherzinger *et al.* [86] present ControVol, a framework controlling schema evolution in NoSQL applications. ControVol statically type checks object mapper class declarations against earlier versions in the code repository. ControVol is capable of warning developers of risky cases of mismatched data and schema. ControVol also suggests and performs automatic fixes to resolve possible schema migration problems.

5.1.3 Schema evolution at runtime

Several approaches and tools exist to support database schema evolution at *runtime*, with the aim to reduce the downtime of data-intensive systems subject to evolution [30]. Among those tools, let us mention Large Hadron Migrator [54], the openark kit [74], the Percona toolkit [76], and TableMigrator [94].

The above tools are limited to applying structural changes to *one* database table at a time. They all rely on a similar schema evolution strategy: a structural copy of the database table under change is created as a *ghost* table. The schema change operation is then applied to this ghost table. Data is copied from the original table to the ghost table and kept synchronized during a transition period using database triggers. Once the copy of

Schema construct	**Semantic impact of construct modification**		
	S^+	S^-	$S^=$
Entity type	add	remove	rename convert to attribute convert to rel. type split/merge
Relationship type	add	remove	rename convert to ent. type convert to attribute
Role	create increase max. card. decrease min. card. add ent. type	delete decrease max. card. increase min. card. remove ent. type	rename
Is-a relationship	add change type	remove change type	
Attribute	add increase max. card. decrease min. card. extend domain change type	remove decrease max. card. increase min. card. restrict domain change type	rename convert to ent. type aggregate disaggregate instantiate concatenate
Identifier	add add component	remove remove component	rename change type
Constraints	add add component change type	remove remove component change type	rename
Access key	add add attribute	remove remove attribute	rename
Collection	add add ent. type	remove remove ent. type	rename

Table 3: Semantic classification of conceptual schema modifications (expressed in the GER model).

Ref.	Atomic Change	Category	DDL (MySQL Implementation)
A1	Add Table	Transformation	CREATE TABLE *t_name*
A2	Add Column	Transformation	ALTER TABLE *t_name* ADD *c_name*
A3	Add View	Transformation	CREATE VIEW *v_name* AS ...
A4	Drop Table	Structure Refactoring	DROP TABLE *t_name*
A5	Rename Table	Structure Refactoring	ALTER TABLE *o_t_name* RENAME *n_t_name*
A6	Drop Column	Structure Refactoring	ALTER TABLE *t_name* DROP COLUMN *c_name*
A7	Rename Column	Structure Refactoring	ALTER TABLE *t_name* CHANGE COLUMN *o_c_name n_c_name*
A8	Change Column Datatype	Structure Refactoring	ALTER TABLE *t_name* MODIFY COLUMN *c_namec_def*
A9	Drop View	Structure Refactoring	DROP VIEW *v_name*
A10	Add Key	Structure Refactoring	ALTER TABLE *t_name* ADD KEY *k_name*
A11	Drop Key	Structure Refactoring	ALTER TABLE *t_name* DROP KEY *k_name*
A12	Add Foreign Key	Referential Integrity Refactoring	ALTER TABLE *t_name* ADD FOREIGN KEY *fk_name* ...
A13	Drop Foreign Key	Referential Integrity Refactoring	ALTER TABLE *t_name* DROP FOREIGN KEY *fk_name*
A14	Add Trigger	Referential Integrity Refactoring	CREATE TRIGGER *trig_name* ... ON TABLE *t_name* ...
A15	Drop Trigger	Referential Integrity Refactoring	DROP TRIGGER *trig_name*
A16	Add Index	Architectural Refactoring	ALTER TABLE *t_name* ADD INDEX *idx_name*
A17	Drop Index	Architectural Refactoring	ALTER TABLE *t_name* DROP INDEX *idx_name*
A18	Add Column Default Value	Data Quality Refactoring	ALTER TABLE *t_name* MODIFY COLUMN *c_name* SET DEFAULT *value*
A19	Drop Column Default Value	Data Quality Refactoring	ALTER TABLE *t_name* MODIFY COLUMN *c_name* DROP DEFAULT
A20	Change Column Default Value	Data Quality Refactoring	ALTER TABLE *t_name* MODIFY COLUMN *c_name* SET DEFAULT *value*
A21	Make Column Not NULL	Data Quality Refactoring	ALTER TABLE *t_name* MODIFY COLUMN *c_name* NOT NULL
A22	Drop Column Not NULL	Data Quality Refactoring	ALTER TABLE *t_name* MODIFY COLUMN *c_name* NULL
A23	Add Stored Procedure	Method Refactoring	CREATE PROCEDURE *pro_name* ...
A24	Drop Stored Procedure	Method Refactoring	DROP PROCEDURE *pro_name*

Figure 11: List of atomic schema modification operators, from [80].

the data has fully completed, the original table and the ghost table atomically switch names, and the original table can be dropped.

The Online Schema Change tool [32] (OSC for short), provided by Facebook, follows a similar approach, but the triggers are used to log the data modification in a separate table, in order to replay them *asynchronously* on the ghost table. Github's gh-ost [35] (Online Schema migration Tool) follows an alternate approach: It reads the binary log of the database, and then replicates data changes to the ghost table asynchronously. These asynchronous approaches are particular suitable to migrate very large amounts of data while keeping an acceptable level of data access performance.

de Jong *et al.* [30] present QuantumDB, a tool-supported approach that support relational database schema evolution at runtime, in the context of continuous deployment. In contrast to related tools discussed above, the QuantumDB approach supports the modification of *multiple* tables at the same time. It also allows *several* active database schemas to *co-exist*, during a certain transition period, thereby making this *hybrid* state fully-transparent from the application programs point of view. This mechanism is essential in order to actually achieve zero-downtime schema evolution, even when applying *blocking* DDL schema change operations to the database.

5.2 Intra-platform database evolution in TYPHON

The intra-platform schema evolution methodology and tools of TYPHON will get inspired from several approaches described above. First, we will benefit from the TyphonML generic and conceptual modeling language being developed in WP2 for representing database schemas, encompassing the actual physical schemas of the underlying hybrid data stores. Similarly to the DB-MAIN approach of Hick and Hainaut [46, 47], we will express our schema evolution operators at this high level of abstraction. The schema evolution operators will rely on the DDL operators provided by TyphonQL to propagate the changes at the backend data stores level. Achieving mixed-state, as done in the QuantumDB approach [30], seems to be promising in the context of TYPHON polystore evolution.

6 Inter-platform database evolution

In this section, we present existing approaches and techniques related to *inter-platform* database evolution scenarios, i.e., scenarios involving the replacement of the database platform. We particularly focus on the migration (and bridging) of relational databases to NoSQL data stores.

6.1 Existing approaches

Relational databases and NoSQL databases rely on different models, the relational algebra model for relational databases and either document-oriented, key value, table-oriented or graph-oriented models for NoSQL databases. Those NoSQL models are briefly introduced in 2.2. Below we summarize state-of-the-art approaches and tools supporting the migration of schemas and data from a relational platform to NoSQL. We first start with general approaches, that aim to remain independent from the specific target NoSQL model. Then we present techniques focusing on data migration towards document-oriented and column-oriented data stores, respectively. We finally present existing tools supporting this migration process.

6.1.1 Migrating relational databases to NoSQL in general

The most naive way to migrate a relational schema to a non-relational model is to follow the so-called *normalization* method. This method relies on a *one-to-one* mapping between the source and target database schemas, e.g., in the case of a relational-to-MongoDB migration, each source table becomes a document in the target document store. This can be seen as the easiest way to migrate a relational schema to NoSQL data model, with the advantage that the resulting schema is, in principle, also normalized. But this method shows several disadvantages too. For instance, most NoSQL data models do not support join queries. This means that join SQL queries have to be split into several NoSQL queries, invoked separately on the target database. This limitation may cause a major loss of data access performance, which is paradoxal in NoSQL. In addition, non-relational platforms generally do not support database transactions. Multiple parallel table updates can therefore fail and lead to data inconsistency. In summary, migrating data using the normalization method, while very simple to achieve, may significantly reduce the expected positive impact of migrating relational data to NoSQL.

Ghotiya *et al.* [34] established a survey of current available techniques and tools for migrating data to NoSQL models. Liang *et al.* [56] proposed a transitional model between relational and NoSQL models. This model is provided with a set of mapping strategies that help to migrate the relational structure and data to the physical schema of the particular target NoSQL model. The main advantage of this approach is that it does not restrict itself to a specific target NoSQL model, any existing ones can be expressed. However, on the other hand, the source application code has to be maintained and evolved accordingly.

An algorithm for schema conversion has been proposed by Zhao *et al.* [107]. This algorithm is based on a graph-based representation of the databases: vertex are tables and edges are relationships between them. Starting from the leaf nodes, the implementation will then build a set of nested sets of tables. This solution results in very efficient query execution. In addition, all queries are directly transposable since every information is present in a single collection, so that no joins are needed. This implementation has as drawback the size of the resulting database. Indeed the total size may grow rapidly depending on the number of foreign keys in the source relational database.

In order to avoid this significant size increase Yoo *et al.* [105] propose a technique for schema migration that is based on *Column-level denormalization*. Denormalization is the process of duplicating data to avoid the need for join queries that cannot be achieve directly in NoSQL systems. But column-level denormalization only

duplicates columns that are accessed in a non primary-foreign key join relationship. Their *Atomic aggregates* method transforms a transaction into an atomic update of multiple tables. An experiment shows that the performance reached compared to state-of-the-art method is two times faster and uses 2.8 times less space.

Instead of migrating data Xu *et al.* [103] propose to build a middleware system that will integrate relational database management systems and NoSQL systems. Such an integration appears very useful for applications that will need to combine both database paradigms instead of using only one of them. Relational database management systems are focused on ACID properties and are therefore more convenient for important small sized databases and write-intensive processes. In contrast, NoSQL focuses on velocity and scalability which is more suitable for big data and read-intensive processes. Xu *et al.* [103] develop a new language, called ZQL, which is platform-independent and allows users to write queries that will eventually be executed either in an relational database or in a NoSQL datastore. The ZQL module translates the given query to the corresponding system and return the results with full transparency for the user.

Following the same idea of integrating both models into application programs, Sellami *et al.* [87] present an API, called *ODBAPI*, which aims to decouple technology-specific database code from the application code. Using their REST API based on a unified model it is possible to use a unique language to perform queries on multiple data store. Metadata handles the mapping of datasources and drivers. Migration of data becomes easier and the application code is platform-independent which makes the work of developers easier as they do not have to learn each data store language. This API is also extensible in order to easily add other data sources by providing an additional mapping in the API model. However this API is currently limited to basic CRUD operations.

Similarly, the SOS (Save Our System) platform [7] also establishes a pivot model between the different NoSQL models in order to decouple the application code from the technical specificities of the underlying NoSQL databases. It is a common programming interface that hides the technical details of three NoSQL databases: Redis as key-value model, HBase as column-oriented model and MongoDB as document-oriented model. This approach has been implemented using those three systems and applied on a Tweeter use case. Unfortunately they did not integrated the relational model in the use case but following a similar idea, it should be possible to add the relational model to this programming interface.

6.1.2 Migrating relational databases to document-oriented data stores

Before starting to migrate data from a source relational model to a target document-oriented model, Zhao *et al.* [106] studied the theoretical possibility of such a migration process. They used the relational algebra and applied it to the MongoDB document-oriented model to check if it could support the same capabilities. They concluded that document store model like MongoDB also supports relational calculus and therefore data can be migrated between the two as they have the same capability set.

NoSQL Layer, proposed by Rocha *et al.* [85] is a framework to migrate data from relational databases to MongoDB. It contains two modules: The *Data migration* module and the *Data Mapping Module*. The data migration module automatically builds an equivalent structure in MongoDB and then migrate the data. The data mapping module is a data access layer; its goal is to act as an interface between the databases and the application programs. Its main feature is the ability to translate each SQL query into an equivalent MongoDB query, and then convert the returned query results in a SQL-compatible format. Therefore, in principle, no program adaptation is required.

Stanescu [91] propose a mapping algorithm to transform a relational database into a MongoDB database. The algorithm carried out this process by transforming tables and relationships, such as 1:1, 1:N and M:N relationships either into an embedding in the document collection or into a relationship between several collections.

The algorithm makes the choice for a given table T by taking into account the number of references from T to other tables and the number of incoming references pointing to table T from other tables. They implemented this algorithm using the meta-data table of MySQL INFORMATION_SCHEMA. A drawback of this technique that we can point out is that it relies on the explicit declaration of foreign key constraints in the source relational schema. However, it has been showed that foreign keys are not always explicitly declared in the database schema but can instead be implicitly managed by the application programs [22].

6.1.3 Migrating relational databases to column-oriented data stores

Migrating data from relational databases to column-oriented model, such as HBase has been explored by Lee and Zheng [53]. The authors reuse the design principles of denormalization and duplication explained before and they add an automatic way to build the row key. Using the MySQL meta data table, they build linked list following the paths foreign keys - primary keys. The longest one is then established as the row key. Their experimental results showed an improvement of 47% in access performance.

Li [55] proposes a two-phase method to migrate data from a relational database to a HBase distributed data store. They provide three guidelines to convert the data automatically under this migration scenario. The first guideline is to group together correlated data and adding foreign keys between different schemas. The second guideline is to minimize the number of foreign keys by merging data together. Finally, the inter-schema mappings are exploited to create a set of queries that transforms the data automatically.

Kim *et al.* [52] also studied the migration towards HBase and its column-oriented NoSQL model. They use Apache Phoenix to migrate their SQL queries. It is a SQL layer working on top of HBase. They considered Phoenix as too basic to support complex queries such as queries with join operations, subqueries or *HAVING* clauses. Therefore they had to manually refactor those queries. In order to fully and correctly support previous SQL queries they recommend to use techniques such as *Denormalization* and *Atomic Aggregates* [105]. Moreover they evaluated all their SQL queries with their respective NoSQL queries in an SVM classifier, which provide some insights about which categories of queries actually translate better in a NoSQL environment.

Liao *et al.* [57] also use HBase and Apache Phoenix in the context of relational to column-oriented data migration. Their idea is to use relational database management systems and NoSQL databases *in combination*. The relational data are replicated into a NoSQL database, and the authors aim to ensure consistency without causing downtime. Application queries do not need to be adapted as the authors make also use of a translator module, called *DB Adapter*. Moreover the approach provides the users with three different modes in order to update the NoSQL database. The first one is a *blocking* mode. This mode is more convenient for batch applications as it is performed offline and thus it does not have to process incoming queries while performing the migration. This is also much faster. The second mode is the *online* mode, that can migrate the data while the application is still running without loss of data. This mode takes much more time to complete. The last mode is a compromise between the two other modes.

Once the data have been migrated it can be useful to verify and validate that the source and target datasets are equivalent, i.e., that the migration was performed without data loss. Goyal *et al.* [37] explores this line of research by exploiting the bloom filters, based on a probabilist technique that can assert the presence/absence of an element in a particular data set.

6.1.4 Supporting tools

A few existing tools can support the process of data migration between relational databases and NoSQL platforms. Those tools have been used in several papers presented in the previous section: Apache Sqoop and Apache Phoenix.

- **Apache Sqoop**[90] is able to transfer bulk data from relational databases to HDFS (compatible with Hadoop/Hive/HBase). Therefore a user can transfer data and perform a MapReduce operation with Hadoop easily. The result can afterwards be exported again in the relational database. When using this tool it can also be interesting to apply the research of Hsu *et al.* [48] where they propose a method to better split the data across nodes and thus improve the read performance of JOIN operations passed to Sqoop.
- **Apache Phoenix**[78] is an SQL layer that can provide data access on top of HBase databases. It allows users to manipulate a NoSQL column-oriented database by means of the SQL language.

6.2 Inter paradigm database evolution in TYPHON

When reading and analyzing the research papers summarized above, one can identify two main approaches to data migration, either through a direct mapping or via an intermediate pivot model. The TYPHON ambition is to combine the advantages of both approaches. TYPHON will use TyphonML as a generic pivot modeling language for representing database schemas and will use TyphonQL as a generic intermediate language for expressing database queries. The platform-independence and extensibility of those modeling and query languages will greatly facilitate database evolutions in general, and more specifically data migration between different database platforms within the hybrid polystores. This is confirmed by several authors who addressed the migration of relational databases to document-oriented or to column-based datastores.

7 Program and query adaptation

The adaptation of client application programs under database schema evolution is a complex process. Most existing tool-supported approaches to this process rely on transformational techniques, generative techniques or a combination of both. For instance, several authors attempt to contain the ripple effect of changes to the database schema, e.g., by generating *wrappers* [98], views or APIs that provide/enable backward compatibility and by transforming the programs in order to interface them with those intermediate layers.

7.1 Existing approaches

7.1.1 Analyzing the impact of database schema evolution

A first step towards program adaptation has been explored by Grolinger and Capretz [39]. They propose the integration of database accesses in the unit tests. They add a layer which effectively accesses the database instead of mocking it. In this way the actual queries can be checked against the (evolving) schema. If needed they also modify the queries in order to query the structure of the schema instead of the actual data, with the aim to increase data access performance. By validating queries to the schema they can identify source code fragments in the programs that have become invalid and that would therefore fail, in order to help programmers when adapting programs to evolutions of the database.

Maule *et al.* [63] propose an impact analysis approach to database schema changes. This approach relies on a combination of program slicing and dataflow analysis techniques. The program slicing techniques aim to reduce the number of lines of code that the program analyzer has to parse. The obtained program subset is then further analyzed through dataflow analysis, in order to extract and collect all possible database queries, together with their respective input parameters. A last step requires the user to provide an hypothetical schema change as input of the tool. The latter then derives the set of impacted queries and produces a detailed impact report. The user can then (manually) adapt the application programs accordingly.

Liu *et al.* [60] propose a novel graph, which they call the *attribute dependency graph*, in order to show the dependencies between attributes in a database application and the programs involved. This approach, implemented for PHP-based applications, extracts the attribute dependency graph from the programs source code by means of inter-procedural static program analysis. The purpose was to provide developers with support to different maintenance tasks, by focusing on impact analysis.

In [33] the authors provide a tool and program slicing technique specifically designed to adapt the programs source code as well as related regression tests when a database schema change occurs. This two-folded impact analysis method aims to identify the source code statements affected by the schema changes and the affected test suites associated to these source code fragments. They implemented their approach for PL/SQL applications.

Chang *et al.* [14] propose a formal framework for database refactoring which is based on a logical model of changes, and that can automatically identify inconsistencies in the application code as well as database modeling problems.

Meurice *et al.* [68] present a tool-supported approach, that allows developers to simulate a database schema change and automatically determine the set of source code locations that would be impacted by this change. Developers are then provided with recommendations about what they should modify at those source code locations in order to avoid query-schema inconsistencies. The approach has been designed to deal with Java systems that use dynamic data access frameworks such as JDBC, Hibernate and JPA.

ControVol, by Scherzinger *et al.* [86] is a framework pluggable into an IDE that can support developers when evolving their source code and specifically their object mapper class declarations, in the particular context of a NoSQL data stores.

7.1.2 Supporting schema-program co-evolution

Cleve *et al.* [18] presented a tool-supported approach that combines the automated derivation of a relational database from a conceptual schema, and the automated generation of a data manipulation API providing programs with a *conceptual* view of the relational database. Schema change is achieved through a systematic transformation process, keeping track of the mapping between the successive versions of the schema. Database schema evolutions are then propagated through API regeneration so that client applications are protected against information-preserving schema changes.

The *PRISM* workbench by Curino *et al.* [29] provides a highly integrated support to *relational* schema evolution. This tool suite includes (1) a language for the specification of Schema Modification Operators (*SMOs*) for relational schemas, (2) impact analysis tools that evaluate the effects of such operators, (3) automatic data migration support, and (4) translation of old queries to work on the new schema. Query adaptation derives from the *SMOs* and combines SQL view generation and query rewriting techniques. Most *SMOs* map directly to an equivalent *DDL* statement, but some *SMOs* support more complex operations on the database schema like splitting or combining two tables in terms of records or columns.

The *2LT* project [102] aims to formalize and to provide generic support for *two-level transformations*, which involve a transformation on the level of types with transformations on the level of values and operations. The solutions offered by the *2LT* project combine existing techniques of data refinement, typed strategic rewriting, point-free program transformation and advanced functional programming. This generic approach revealed to be applicable to the coupled transformation of database schemas, data instances, queries, and constraints.

Bidirectional transformations [95] can also be used to decouple the evolution of the database schema from the evolution of the queries, by allowing changes to the schema to be implemented while some queries can remain unchanged. Terwilliger *et al.* [96] introduced the concept of *Channel* to formalize transformations that translate application code queries to a "virtual" database schema to equivalent queries into the actual schema.

Stonebraker *et al.* [93] discuss two possible ways of co-evolving database schemas and application programs. A first way is the *data-first* way, the one recommended by good practices in database designs. It consists of first evolving the database schema, keeping it in the third normal form (3NF) and then adapting the application program regarding those changes. In real-world companies this is almost never applied [92], due to potential higher cost and difficulties of program maintenance. Therefore the *application-first* strategy is favoured. It consists of mitigating or even avoiding application code changes. This discourages database administrators to try to modify the existing structure of the database. Instead, they rather continuously *add* fields or tables. Those two approaches do not constitute fully-satisfying solutions. The first one leads to program decay as applications may not correctly be adapted to the schema changes. The second leads to a database decay as data may be duplicated and thus the schema may become less and less conform to the 3NF. To avoid such problems, Stonebraker *et al.* recommend to add an intermediate layer accessing the database(s), and to make application programs manipulate data through this layer. In case of schema evolutions, the database access API would not change from the programs' point of view, but only the implementation of the API functions would change, under the responsibility of the database administrators. This access layer can be implemented, for example, via a REST API.

7.1.3 Analyzing schema-program co-evolution

Some researchers have also *observed* how databases and programs co-evolve over time. Lin *et al.* [59] study the so-called *collateral* evolution of applications and databases, in which the evolution of an application is separated from the evolution of its persistent data, or from the database. They investigated how application programs and database management systems in popular open source systems (Mozilla, Monotone) cope with database schema changes and database format changes. They observed that collateral evolution can lead to potential problems. However, the number of schema changes reported is very limited. In Mozilla, 20 table creations and 4 table deletions are reported in a period of 4 years. During 6 years of Monotone schema evolution, only 9 tables were added while 8 tables were deleted.

Qiu *et al.* [80] conduct a large-scale empirical study on ten popular database applications from various domains to analyze how schemas and application code co-evolve. In particular, they study the evolution histories from the respective repositories to understand whether database schemas evolve frequently and significantly, how schemas evolve and impact the application code. In their approach, the authors try to estimate the impact of a database schema change in the code. This estimation is performed with a simple difference extractor calculating changed source lines between two versions.

Karahasanoić [51] studied how the maintenance of application consistency can be supported by identifying and visualizing the impacts of changes in evolving object-oriented systems, including changes originating from a database schema. This work focuses on the evolution of object-oriented databases.

Goeminne *et al.* [36] study the co-evolution between code-related and database-related activities in data-intensive systems combining several ways to access the database (native SQL queries and Object-Relational Mapping). They empirically analyzed the evolution of the usage of SQL, Hibernate and JPA in a large and complex open source information system. Interestingly, they observed that the practice of using embedded SQL queries is still common today.

Meurice and Cleve[67] provide a method analyzing the joint evolution of the application programs and their underlying NoSQL data store. They use the application code and its evolution history to identify implicit changes in the data structure and potential points of failure due to this schema evolution.

7.2 Program and query adaptation in TYPHON

The approaches described above will be definitely inspiring in the context of the TYPHON project, in particular regarding the way (evolving) hybrid polystores will be accessed by client applications through the TyphonQL query language and the Typhon access API. As done by several authors [96, 29], we will define and implement a set of co-evolution rules between TyphonML schemas and related TyphonQL queries. The regeneration of the TYPHON API following a schema change will also greatly facilitate the co-evolution of the database schema and the external application programs, as shown in the approach of Cleve et al [18]. The different studies analyzing schema-program co-evolution over time provide us with important insights about the most frequent schema evolution scenarios and operators, as well as the main difficulties encountered by developers as far as program adaptation is concerned.

8 Summary

In this section we present a comparative summary table of the approaches detailed in the intra-platform database evolution, inter-platform database evolution and program adaptation sections. Table 4 characterizes the different surveyed approaches based on different dimensions introduced in Section 2. We first indicate whether the approach focuses on the evolution processes described in 2.5, namely *impact analysis, database change* and *program change*.

Then, the semantics-related columns indicate if the studied approach considers evolution scenarios involving, semantics-decreasing, semantics-increasing or semantics-preserving schema changes.

The approach dimension aims to explain how the approaches actually achieve the database evolution processes that they support.

The last two columns refer to the source and target database management system considered.

For the program adaptation approaches, the source column refers to the kind of input required by the approach. The output is left blank when it consists of an impact analysis report or a set of change recommendations in the program.

Reference	Process			Semantic			Approach	Source	Target
	Impact analysis	Database change	Program change	S^+	S^-	$S^=$			
Intra-platform evolution									
Sjøberg [88]	X			X	X	X	Empirical study	relational DB	relational DB
Curino *et al.* [28]	X			X	X	X	Empirical study	relational DB	relational DB
Vassiliadis *et al.* [99, 100]	X			X	X	X	Empirical study	relational DB	relational DB
Cleve *et al.* [19]		X		X	X	X	History analysis	relational DB	relational DB
Hick and Hainaut [46, 47]		X		X	X	X	Intermediary model	relational DB	relational DB
Scherzinger [86]		X	X	X	X	X	Intermediary tool	MongoDB	MongoDB
Lin and Neamtiu [58]		X	X	X	X	X	Empirical study	relational DB	relational DB
Curino [27]		X	X	X	X	X	Intermediary tool	relational DB	relational DB
Qiu *et al.* [80]	X			X	X	X	Empirical study	relational DB	relational DB
Inter-platform evolution									
Liang *et al.* [56]	X	X				X	Intermediary model	relational DB	All NoSQL models
Zhao *et al.* [107]		X				X	Intermediary model	relational DB	All NoSQL models
Yoo *et al.* [105]		X				X	Intermediary model	relational DB	All NoSQL models
Xu *et al.* [103]		X	X	X	X	X	Intermediary model	relational DB, NoSQL	relational DB, NoSQL
Sellami *et al.* [87]		X	X	X	X	X	Intermediary model	Generic	Generic
Atzeni [7]							Intermediary model	Redis, HBase, MongoDB	Redis, HBase, MongoDB
Zhao *et al.* [106]	X					X	Relational algebra	relational DB	MongoDB
Rocha *et al.* [85]		X	X	X	X	X	Direct mapping	relational DB	MongoDB
Stanescu [91]		X				X	Direct mapping	MySQL	MongoDB
Li [55]		X				X	Direct mapping	relational DB	HBase
Lee and Zheng [53]		X				X	Direct mapping	relational DB	HBase
Kim *et al.* [52]		X	X				Intermediary software	relational DB	HBase
Liao *et al.* [57]		X					Intermediary software	relational DB, HBase	HBase
Apache Sqoop		X				X	Intermediary tool	relational DB, Hive, HBase	Hive, HBase, relational DB
Apache Phoenix			X	X	X	X	Intermediary tool	relational DB, HBase	relational DB, HBase
Program adaptation									
Thiran *et al.* [98]			X	X	X	X	Intermediary model	relational DB	relational DB
Cleve *et al.* [18]		X	X	X	X	X	Intermediary model	relational DB	relational DB
Visser [102]		X	X	X	X	X	Theoretical formalization	relational DB	relational DB
Terwilliger *et al.* [95]		X	X	X	X	X	Theoretical formalization	relational DB	relational DB
Grolinger and Capretz [39]	X			X	X	X	Intermediary tool	relational DB	relational DB
Meurice *et al.* [68]	X		X	X	X		Intermediary tool	JDBC, ORM	JDBC, ORM
Maule *et al.* [63]	X			X	X	X	Program slicing	Application code	-
Liu *et al.* [60]	X		X	X	X		Dependency graph	PHP application	-
Gardikiotis *et al.*[33]	X		X	X	X		Program slicing	PL/SQL	-
Chang *et al.* [14]		X	X	X	X	X	Intermediary model		
Stonebraker *et al.* [93]			X	X	X	X	Intermediary model	Generic	Generic
Meurice and Cleve[67]			X	X	X		Intermediary tool	MongoDB java application code	-
Lin *et al.* [59]	X						Empirical study	relational DB, Application code	-
Qiu *et al.* [80]	X			X	X	X	History evolution	relational DB, Application code	-
Karahasanoić [51]	X						Visualization	Object oriented databases	-
Goeminne *et al.* [36]	X							Application code	-
TYPHON	X	X	X	X	X	X	Intermediary model	Generic	Generic

Table 4: Comparison of database evolution and program adaptation approaches.

9 Conclusions

In this document, we identified, presented and compared the current state-of-the-art approaches to database evolution in general, and to database migration in particular. A large set of approaches and tools have been proposed, mainly in the context of evolving *a single* database at a time. The challenges of the TYPHON project will be to evolve hybrid polystores as well as related database queries in a joint and consistent manner. To achieve that, TYPHON will rely on generative and co-transformational techniques and will benefit from the whole TYPHON approach including (1) TyphonML, a generic, high-level modeling language for database schemas; (2) TyphonQL, a meta-query language compiled towards concrete query languages for relational and NoSQL databases, (3) the automatically-generated Typhon API providing developers call-level interface to the hybrid polystores and (4) TyphonDL, allowing to (re)deploy the evolving hybrid polystores on physical or virtual machines.

References

[1] Fatma Abdelhedi, Amal Ait Brahim, Faten Atigui, and Gilles Zurfluh. Processus de transformation mda d'un schéma conceptuel de données en un schéma logique nosql. In *34e Congrès Informatique des Organisations et Systèmes d'Information et de Décision (INFORSID 2016)*, 2016.

[2] Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. WAFA: fine-grained dynamic analysis of web applications. In *Proceedings of the 11th IEEE International Symposium on Web Systems Evolution, WSE 2009, 25-26 September 2009, Edmonton, Alberta, Canada*, pages 141–150, 2009.

[3] Scott W. Ambler and Pramodkumar J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley, 2006.

[4] Arangodb. `https://www.arangodb.com`.

[5] Paolo Atzeni. Data modelling in the nosql world: A contradiction? In *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016*, CompSysTech '16, pages 1–4, New York, NY, USA, 2016. ACM.

[6] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. Data modeling in the nosql world. *Computer Standards & Interfaces*, pages –, 2016.

[7] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi. Uniform access to non-relational database systems: The sos platform. In *International Conference on Advanced Information Systems Engineering*, pages 160–174. Springer, 2012.

[8] Carol Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design : An Entity-Relationship Approach*. Benjamin/Cummings, 1992.

[9] A. Behm, A. Geppert, and K.R. Dittrich. On the migration of relational schemas and data to object-oriented database systems. In *Proceedings of Re-Technologies in Information Systems*, Klagenfurt, Austria, December 1997.

[10] Alessandro Bianchi, Danilo Caivano, and Giuseppe Visaggio. Method and process for iterative reengineering of data in a legacy system. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 86–, 2000.

[11] Michael L. Brodie and Michael Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann, 1995.

[12] Francesca Bugiotti, Luca Cabibbo, Paolo Atzeni, and Riccardo Torlone. Database design for nosql systems. In *Proc. of the 33rd International Conference on Conceptual Modeling (ER 2014)*, volume 8824 of *Lecture Notes in Computer Science*, pages 223–231. Springer, 2014.

[13] Cassandra. `http://cassandra.apache.org/`.

[14] S.-K. Chang, V. Deufemia, G. Polese, and M. Vacca. A logic framework to support database refactoring. In *Proc. of DEXA'07*, pages 509–518. Springer, 2007.

[15] Roger H. L. Chiang, Terence M. Barron, and Veda C. Storey. Reverse engineering of relational databases: extraction of an eer model from a relational database. *Data Knowl. Eng.*, 12(2):107–142, 1994.

[16] Elliot J. Chikofsky. *"The Necessity of Data Reverse Engineering". Foreword for Peter Aiken's Data Reverse Engineering*, chapter 0, pages 8–11. McGraw Hill, 1996.

[17] Cleve. *Program analysis and transformation for data-intensive systems evolution*. PhD thesis, University of Namur, 2009.

[18] Anthony Cleve, Anne-France Brogneaux, and Jean-Luc Hainaut. A conceptual approach to database applications evolution. In *Proc. of ER 2010*, volume 6412 of *LNCS*, pages 132–145. Springer, 2010.

[19] Anthony Cleve, Maxime Gobert, Loup Meurice, Jerome Maes, and Jens Weber. Understanding database schema evolution: A case study. *Science of Computer Programming*, 97:113–121, 2015.

[20] Anthony Cleve and Jean-Luc Hainaut. *Dynamic Analysis of SQL Statements for Data-Intensive Applications Reverse Engineering*, pages 192–196. IEEE Computer Society Press, 2008.

[21] Anthony Cleve, Jean Henrard, and Jean-Luc Hainaut. Data reverse engineering using system dependency graphs. In *Proc. of the 13th Working Conference on Reverse Engineering (WCRE'2006)*, pages 157–166, Washington, DC, USA, 2006. IEEE Computer Society.

[22] Anthony Cleve, Jean-Roch Meurisse, and Jean-Luc Hainaut. Database semantics recovery through analysis of dynamic SQL statements. *Journal on Data Semantics*, 15:130–157, 2011.

[23] Anthony Cleve, Nesrine Noughi, and Jean-Luc Hainaut. Dynamic program analysis for database reverse engineering. In *GTTSE*. Springer. LNCS, 2012.

[24] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[25] Couchbase. `https://www.couchbase.com/`.

[26] Couchdb. `http://couchdb.apache.org/`.

[27] C. A. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Automating the database schema evolution process. *The VLDB Journal*, 22(1):73–98, February 2013.

[28] Carlo A Curino, Hyun J Moon, Letizia Tanca, and Carlo Zaniolo. Schema Evolution In Wikipedia. In *Proc. of ICEIS 2008*, pages 323–332, 2008.

[29] Carlo A. Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Update rewriting and integrity constraint maintenance in a schema evolution support system: Prism++. *Proc. VLDB Endow.*, 4(2):117–128, November 2010.

[30] Michael de Jong, Arie van Deursen, and Anthony Cleve. Zero-downtime sql database schema evolution for continuous deployment. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 143–152, Piscataway, NJ, USA, 2017. IEEE Press.

[31] Giuseppe Antonio Di Lucca, Anna Rita Fasolino, and Ugo de Carlini. Recovering class diagrams from data-intensive legacy systems. In *Proc. of the 16th IEEE International Conference on Software Maintenance (ICSM'2000)*, page 52. IEEE Computer Society, 2000.

[32] Facebook. Online schema change (osc). https://github.com/facebookincubator/OnlineSchemaChange.

[33] Spyridon K Gardikiotis and Nicos Malevris. A two-folded impact analysis of schema changes on database applications. *International Journal of Automation and Computing*, 6(2):109–123, 2009.

[34] Sunita Ghotiya, Juhi Mandal, and Saravanakumar Kandasamy. Migration from relational to nosql database. volume 263 of *IOP Conference Series: Materials Science and Engineering*, page 042055. IOP Publishing, 2017.

[35] Github. Online schema migration (gh-ost). http://githubengineering.com/gh-ost-github-s-online-migration-tool-for-mysql/.

[36] Mathieu Goeminne, Alexandre Decan, and Tom Mens. Co-evolving code-related and database-related changes in a data-intensive software system. In *CSMR-WCRE '14*, pages 353–357, 2014.

[37] A. Goyal, A. Swaminathan, R. Pande, and V. Attar. Cross platform (rdbms to nosql) database validation tool using bloom filter. In *2016 International Conference on Recent Trends in Information Technology (ICRTIT)*, pages 1–5, April 2016.

[38] Graphdb. http://graphdb.ontotext.com/.

[39] Katarina Grolinger and Miriam AM Capretz. A unit test approach for database schema evolution. *Information and Software Technology*, 53(2):159–170, 2011.

[40] Concettina Del Grosso, Massimiliano Di Penta, and Ignacio García Rodríguez de Guzmán. An approach for mining services in database oriented applications. In *11th European Conference on Software Maintenance and Reengineering, Software Evolution in Complex Software Intensive Systems, CSMR 2007, 21-23 March 2007, Amsterdam, The Netherlands*, pages 287–296, 2007.

[41] Jean-Luc Hainaut. A generic entity-relationship model. In *Proceedings of the IFIP WG 8.1 Conference on Information System Concepts: an in-depth analysis*, pages 109–138. North-Holland, 1989.

[42] Jean-Luc Hainaut, Anthony Cleve, Jean Henrard, and Jean-Marc Hick. Migration of legacy information systems. In Tom Mens and Serge Demeyer, editors, *Software Evolution*, pages 105–138. Springer, 2008.

[43] Jean-Luc Hainaut, Jean Henrard, Vincent Englebert, Didier Roland, and Jean-Marc Hick. *Database Reverse Engineering*, pages 263–263. Springer, 2009.

[44] Hbase. https://hbase.apache.org/.

[45] Robin Hecht and Stefan Jablonski. Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341. IEEE, 2011.

[46] Jean-Marc Hick. *Evolution d'applications de bases de données relationnelles - Méthodes et outils*. PhD thesis, University of Namur, 2001.

[47] Jean-Marc Hick and Jean-Luc Hainaut. Database application evolution: A transformational approach. *Data & Knowledge Engineering*, 59:534–558, December 2006.

[48] Jen Chun Hsu, Ching Hsien Hsu, Shih Chang Chen, and Yeh Ching Chung. Correlation aware technique for sql to nosql transformation. In *Ubi-Media Computing and Workshops (UMEDIA), 2014 7th International Conference on*, pages 43–46. IEEE, 2014.

[49] J.-H. Jahnke and J. P. Wadsack. Varlet: Human-centered tool support for database reengineering. In *Proc. Working Conf. Reverse Engineering (WCRE)*, May 1999.

[50] M. A. Jeusfeld and U. A. Johnen. An executable meta model for re-engineering of database schemas. In *Proc. Conf. on the Entity-Relationship Approach*, Manchester, December 1994.

[51] A. Karahasanović. *Supporting Application Consistency in Evolving Object-Oriented Systems by Impact Analysis and Visualisation*. PhD thesis, University of Oslo, 2002.

[52] Ho-Jun Kim, Eun-Jeong Ko, Young-Ho Jeon, and Ki-Hoon Lee. Techniques and guidelines for effective migration from rdbms to nosql. *The Journal of Supercomputing*, pages 1–15, 2018.

[53] C. H. Lee and Y. L. Zheng. Automatic sql-to-nosql schema transformation over the mysql and hbase databases. In *2015 IEEE International Conference on Consumer Electronics - Taiwan*, pages 426–427, June 2015.

[54] Large hadron migrator.

[55] Chongxin Li. Transforming relational database into hbase: A case study. In *2010 IEEE International Conference on Software Engineering and Service Sciences*, pages 683–687, July 2010.

[56] Dongzhao Liang, Yunzhen Lin, and Guiguang Ding. Mid-model design used in model transition and data migration between relational databases and nosql databases. In *Smart City/SocialCom/SustainCom (SmartCity), 2015 IEEE International Conference on*, pages 866–869. IEEE, 2015.

[57] Ying-Ti Liao, Jiazheng Zhou, Chia-Hung Lu, Shih-Chang Chen, Ching-Hsien Hsu, Wenguang Chen, Mon-Fong Jiang, and Yeh-Ching Chung. Data adapter for querying and transformation between sql and nosql database. *Future Generation Computer Systems*, 65:111 – 121, 2016. Special Issue on Big Data in the Cloud.

[58] Dien-Yen Lin and Iulian Neamtiu. Collateral evolution of applications and databases. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 31–40. ACM, 2009.

[59] Dien-Yen Lin and Iulian Neamtiu. Collateral evolution of applications and databases. In *Joint Int'l Workshop on Principles of software evolution and ERCIM software evolution workshop*, pages 31–40. ACM, 2009.

[60] K. Liu, H. B. K. Tan, and X. Chen. Aiding maintenance of database applications through extracting attribute dependency graph. *J. Database Manage.*, 24(1):20–35, January 2013.

[61] Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovering interesting inclusion dependencies: application to logical database tuning. *Inf. Syst.*, 27(1):1–19, 2002.

[62] V. M. Markowitz and J. A. Makowsky. Identifying extended entity-relationship object structures in relational schemas. *IEEE Trans. Softw. Eng.*, 16(8):777–790, 1990.

[63] Andy Maule, Wolfgang Emmerich, and David Rosenblum. Impact analysis of database schema changes. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 451–460. IEEE, 2008.

[64] Andreas Meier. Providing database migration tools - a practitioner's approach. In *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pages 635–641, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[65] Andreas Meier, Rolf Dippold, Jacky Mercerat, Alex Muriset, Jean-Claude Untersinger, Robert Eckerlin, and Flavio Ferrara. Hierarchical to relational database migration. *IEEE Software*, 11(3):21–27, 1994.

[66] K. Menhoudj and M. Ou-Halima. Migrating data-oriented applications to a relational database management system. In *Proc. Int'l Workshop on Advances in Databases and Information Systems*, Moscow, 1996.

[67] L. Meurice and A. Cleve. Supporting schema evolution in schema-less nosql data stores. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 457–461, Feb 2017.

[68] Loup Meurice, Csaba Nagy, and Anthony Cleve. Detecting and preventing program inconsistencies under database schema evolution. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, pages 262–273. IEEE, 2016.

[69] Loup Meurice, Fco Javier Bermúdez Ruiz, Jens H. Weber, and Anthony Cleve. *Establishing referential integrity in legacy information systems - Reality bites!*, pages 461–465. Institute of Electrical and Electronics Engineers Inc., 12 2014.

[70] R. Missaoui, R. Godin, and H. Sahraoui. Migrating to an object-oriented databased using semantic clustering and transformation rules. *Data Knowledge Engineering*, 27(1):97–113, 1998.

[71] Mongodb. `https://www.mongodb.com/`.

[72] Shamkant B. Navathe and A. M. Awong. Abstracting relational and hierarchical data with a semantic data model. In *Proc. of the Sixth International Conference on Entity-Relationship Approach (ER'1987)*, pages 305–333. North-Holland Publishing Co., 1988.

[73] Neo4j. `https://neo4j.com/`.

[74] Shlomi Noach. The openark kit. `http://code.openark.org/forge/openark-kit`.

[75] Nattapon Pannurat, Nittaya Kerdprasop, and Kittisak Kerdprasop. Database reverse engineering based on association rule mining. *CoRR*, abs/1004.3272, 2010.

[76] The percona toolkit. `http://www.percona.com/software/percona-toolkit`.

[77] Jean-Marc Petit, Jacques Kouloumdjian, Jean-Francois Boulicaut, and Farouk Toumani. Using queries to improve database reverse engineering. In *Proc. of the 13th International Conference on the Entity-Relationship Approach (ER'1994)*, pages 369–386. Springer-Verlag, 1994.

[78] Apache phoenix. `https://phoenix.apache.org`.

[79] William J. Premerlani and Michael R. Blaha. An approach for reverse engineering of relational databases. *Commun. ACM*, 37(5):42–ff., 1994.

[80] Dong Qiu, Bixin Li, and Zhendong Su. An empirical analysis of the co-evolution of schema and code in database applications. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 125–135. ACM, 2013.

[81] E. Rahm and P. A. Bernstein. An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31, December 2006.

[82] Ravi Ramdoyal, Anthony Cleve, and Jean-Luc Hainaut. Reverse engineering user interfaces for interactive database conceptual analysis. In *Proc. of CAiSE'10*, volume 6051 of *LNCS*. Springer, 2010.

[83] Redis. `https://redis.io/`.

[84] Riak kv. `http://basho.com/products/riak-kv/`.

[85] Leonardo Rocha, Fernando Vale, Elder Cirilo, Dárlinton Barbosa, and Fernando Mourão. A framework for migrating relational datasets to nosql1. *Procedia Computer Science*, 51:2593–2602, 2015.

[86] S. Scherzinger, T. Cerqueus, and E. C. d. Almeida. Controvol: A framework for controlled schema evolution in nosql application development. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1464–1467, April 2015.

[87] Rami Sellami, Sami Bhiri, and Bruno Defude. Odbapi: a unified rest api for relational and nosql data stores. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 653–660. IEEE, 2014.

[88] Dag Sjøberg. Quantifying schema evolution. *Info. & Softw. Techn.*, 35(1):35 – 44, 1993.

[89] I. Skoulis, P. Vassiliadis, and A. Zarras. Open-source databases: Within, outside, or beyond lehman's laws of software evolution? In *CAISE '14*, volume 8484 of *LNCS*, pages 379–393. Springer, 2014.

[90] Apache sqoop. `http://sqoop.apache.org`.

[91] Liana Stanescu, Marius Brezovan, and Dumitru Dan Burdescu. Automatic mapping of mysql databases to nosql mongodb. *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 837–840, 2016.

[92] Michael Stonebraker, Dong Deng, and Michael L Brodie. Database decay and how to avoid it. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 7–16. IEEE, 2016.

[93] Michael Stonebraker, Dong Deng, and Michael L Brodie. Application-database co-evolution: A new design and development paradigm. *New England Database Day*, pages 1–3, 2017.

[94] Table migrator. `https://github.com/freels/table_migrator`.

[95] James Terwilliger, Anthony Cleve, and Carlo Curino. How clean is your sandbox? : Towards a unified theoretical framework for incremental bidirectional transformations. In *Proc. of ICMT 2012*, volume 7307 of *LNCS*, pages 1–23. Springer, 2012.

[96] James F. Terwilliger, Lois M. L. Delcambre, David Maier, Jeremy Steinhauer, and Scott Britell. Updatable and evolvable transforms for virtual databases. *PVLDB*, 3(1):309–319, 2010.

[97] James F. Terwilliger et al. The user interface is the conceptual model. In *Proc. of ER'06*, volume 4215 of *LNCS*, pages 424–436. Springer, 2006.

[98] Philippe Thiran, Jean-Luc Hainaut, Geert-Jan Houben, and Djamal Benslimane. Wrapper-based evolution of legacy information systems. *ACM Trans. Softw. Eng. Methodol.*, 15(4):329–359, 2006.

[99] Panos Vassiliadis, Apostolos V. Zarras, and Ioannis Skoulis. How is life for a table in an evolving relational schema? Birth, death and everything in between. In *Int'l Conf. Conceptual Modeling*, pages 453–466, 2015.

[100] Panos Vassiliadis, Apostolos V. Zarras, and Ioannis Skoulis. Gravitating to rigidity: Patterns of schema evolution – and its absence – in the lives of tables. *Information Systems*, 63:24 – 46, 2017.

[101] Giuseppe Visaggio. Ageing of a data-intensive legacy system: symptoms and remedies. *Journal of Software Maintenance*, 13(5):281–308, 2001.

[102] Joost Visser. Coupled transformation of schemas, documents, queries, and constraints. *Electr. Notes Theor. Comput. Sci.*, 200(3):3–23, 2008.

[103] Jie Xu, Mengji Shi, Chaoyuan Chen, Zhen Zhang, Jigao Fu, and Chi Harold Liu. Zql: A unified middleware bridging both relational and nosql databases. In *Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2016 IEEE 14th Intl C*, pages 730–737. IEEE, 2016.

[104] Hong Yao and Howard J. Hamilton. Mining functional dependencies from data. *Data Min. Knowl. Discov.*, 16(2):197–219, 2008.

[105] Jaejun Yoo, Ki-Hoon Lee, and Young-Ho Jeon. Migration from rdbms to nosql using column-level denormalization and atomic aggregates. *Journal of Information Science & Engineering*, 34(1), 2018.

[106] G. Zhao, W. Huang, S. Liang, and Y. Tang. Modeling mongodb with relational model. In *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, pages 115–121, Sept 2013.

[107] G. Zhao, Q. Lin, L. Li, and Z. Li. Schema conversion model of sql database to nosql. In *2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 355–362, Nov 2014.