Theory and Implementation of a Variational Database Management System

Parisa S. Ataei
Thesis Proposal
Submitted November 3rd, 2020

Abstract

Many problems require working with data that varies in its structure and content. Likewise, many tools and techniques have been developed for dealing with variation in databases with respect to time (e.g., work on database evolution) or space (e.g., work on data integration). However, these specialized approaches neither cover all data variation needs nor provide a solution to deal with database variation both in time and space simultaneously. In this research, we propose a generic framework that considers *variation* orthogonal to relational databases. We extend the relational database theory to incorporate variation explicitly in databases and the query language: we define *variational schemas* for describing variation in the structure of a database *variational queries* for expressing variation in information needs, and *variational databases* for capturing variation in content. Although the model underlying variational database is simple, encoding variation explicitly in databases introduces complexity akin to using preprocessing directives in software. We evaluate the feasibility of this approach by systematically developing two case studies that illustrate how different kinds of variation needs can be encoded and integrated in a variational database and how the corresponding information needs can be expressed as variational queries. We also design and implement a variational database management system as an abstraction layer over a traditional relational database. We demonstrate the applicability and feasibility of our approach on our two case-studies.

1 Introduction

Variation in databases arises when multiple database instances conceptually represent the same database, but, differ slightly either in their schema and/or content. Managing variation in databases is a perennial problem in database literature and appears in different forms and contexts. Specific cases of this problem has been extensively studied including schema evolution [], data integration [], and database versioning [], where each instance has a context-specific solution that is hard-wired to the constrained problem definition. While schema evolution approaches deal with variation in the schema (i.e., the structure of the data) database versioning and data integration systems manage variation in the content of the database. Yet, the database community does not consider them as instances of the same problem.

Consider schema evolution which is an instance of schematic variation in databases that is well-supported [19, 6, 4, 25, 21]. Changes applied to the schema over time are *variation* in the database and every time the database evolves, a new *variant* is generated. Current solutions addressing schema evolution rely on temporal nature of schema evolution by using timestamps [19, 6, 4, 25] or keeping an external file of time-line history of changes applied to the database [21]. These approaches only consider variation in time

and do not incorporate the time-based changes into the database directly, rather they *simulate* the effect of these changes, resulting in brittle systems.

Database-backed software produced by software product line (SPL) is an example where variation arises in databases and is poorly supported. SPL is an approach to developing and maintaining software-intensive systems in a cost-effective, easy to maintain manner by accommodating variation in the software that is being reused. In SPL, a common codebase is shared and used to produce products w.r.t. a set of selected (enabled) features [7]. Different products of a SPL typically have different sets of enabled features or are tailored to run in different environments. These differences impose different data requirements which creates variation in space in the shared database used in the common codebase. The variation is in the form of exclusion/inclusion of tables/attributes based on selected features for a product [**?**]. In practice, software systems produced by a SPL are accommodated with a database that has all attributes and tables available for all variants– a database with universal schema [**?**]. Unfortunately, this approach is inefficient, error-prone, and filled with lots of null values since not all attributes and tables are valid for all variant products. A possible solution to this could be defining views on the universal database per software variant and write queries for each variant against its view [**?**]. However, this is burdensome, expensive, and costly to maintain since it requires developers to generate and maintain numerous view definitions in addition to manually generating and managing the mappings between views and the universal schema for each product.

Additionally, the software product line (SPL) community has realized that variation in software development travels to its artifact including the database. SPL is an approach to developing and maintaining software-intensive systems in a cost-effective, easy to maintain manner by accommodating variation in the software that is being reused. In SPL, a common codebase is shared and used to produce products w.r.t. a set of selected (enabled) features [7]. Different client's products of a SPL typically have different sets of enabled features or are tailored to run in different environments. These differences impose different data requirements which creates variation in space in the shared database used in the common codebase. The SPL community has closed the gap of variation appearing in database schema while developing software and the database schema used in client's application by encoding variation explicitly in data model, called *variable data model* []. However, the direct encoding of variation has not been extended to the database itself by these approaches.

To the best of our knowledge, there is no generic solution that manages all possible kinds of variation in databases. Thus, we explore the idea of considering variation as a *first-class citizen* in databases. Such exploration poses questions such as: How can variation be represented in a generic expressive manner? How can variation be encoded explicitly in databases? What are the benefits and drawbacks of explicitly encoding variation in databases? How applicable and feasible it is to encode variation directly in databases?

To answer these research questions we propose the following research goals:

- Objective 1: Identify the kinds of variation existing in relational databases in different application domains.
- Objective 2: Design a query language and implement a database management system that accommodate variations identified in objective 1.
- Objective 3: Demonstrate how the proposed system can be used to manage variation in databases in different application domains.
- Objective 4: Mechanize proofs of properties of the language and the system.

Table 1: Introduced notations and terminologies with their corresponding section(s).

Name	Notation	Section
Feature	f	
Feature expression	e	
Annotated element x by e	x^e	
Configuration	c	Section 4.1.1
Evaluation of e under c	$\mathbb{E}[\![e]\!]_c$	
Presence condition of entity x	$pc(x)$	
Optional attribute	a	
Variational attribute set	A	Section **??**
Variational relation schema	s	
Variational schema	S	
Variational tuple	u	
Variational relation content	U	Section **??**
Variational table	$t = (s, U)$	
Choice	$e\langle x, y\rangle$	
Variational condition	θ	Section **??**
Variational query	q	

2 Statement of Thesis

The goal of this research is to provide a query language(s) and a database management system that explicitly account for different kinds of variation in relational databases to relieve some of programmer/DBA's labor by providing some guarantees within the query language that accounts explicitly for variation.

3 Preliminaries

In this section, we introduce concepts and notations that we use throughout the paper. Table 1 provides a short overview and is meant as an aid to find definitions faster. Throughout the proposal, we discuss relational concepts and their variational counterparts. For clarity, when we need to emphasize an entity is not variational we underline it, e.g., $\underline{x}$ is a non-variational entity while x is its variational counterpart, if it exists.

3.1 Relational Databases and Relational Algebra

A relational database $\underline{D}$ stores information in a structured manner by forcing data to conform to a *schema* $\underline{S}$ that is a finite set $\{\underline{s}_1, \ldots, \underline{s}_n\}$ of *relation schemas*. A relation schema is defined as $\underline{s} = r(\underline{a}_1, \ldots, \underline{a}_k)$ where each $\underline{a}_i$ is an *attribute* contained in a relation named r. $rel(\underline{a})$ returns the relation that contains the attribute. $type(\underline{a})$ returns the *type* of values associated with attribute $\underline{a}$.

The content of database $\underline{D}$ is stored in the form of *tuples*. A tuple $\underline{u}$ is a mapping between a list of relation schema attributes and their values, i.e., $\underline{u} = (\underline{v}_1, \ldots, \underline{v}_k)$ for the relation schema $r(\underline{a}_1, \ldots, \underline{a}_k)$. Hence a *relation content*, $\underline{U}$, is a set of tuples $\{\underline{u}_1, \ldots, \underline{u}_m\}$. $att(\underline{v})$ returns the attribute the value corresponds to. A *table* $\underline{t}$ is a pair of relation content and relation schema. A *database instance*, $\underline{\mathscr{I}}$, of the database $\underline{D}$ with the schema $\underline{S}$, is a set of tables $\{\underline{t}_1, \ldots, \underline{t}_n\}$. For brevity, when it is clear from the context we refer to a database

$$\underline{\theta} \in \underline{\Theta} \quad ::= \quad \texttt{true} \mid \texttt{false} \mid a \bullet k \mid a \bullet a \mid \neg \underline{\theta} \mid \underline{\theta} \vee \underline{\theta}$$

$$\begin{array}{lcll} \underline{q} \in \underline{\mathbf{Q}} & ::= & \underline{r} & \textit{Relation reference} \\ & \mid & \rho_{\underline{r}} \underline{q} & \textit{Renaming} \\ & \mid & \pi_{\underline{A}} \underline{q} & \textit{Projection} \\ & \mid & \sigma_{\underline{\theta}} \underline{q} & \textit{Selection} \\ & \mid & \underline{q} \bowtie_{\underline{\theta}} \underline{q} & \textit{Join} \end{array}$$

Figure 1: Syntax of relational algebra, where $\bullet$ ranges over comparison operators ($<, \leq, =, \neq, >, \geq$), k over constant values, a over attribute names, and $\underline{A}$ over lists of attributes. The syntactic category $\underline{\theta}$ is relational conditions, and $\underline{q}$ is relational algebra terms.

instance by *database*.

Relational algebra allows users to query a relational database [1]. The first five constructs are adapted from relational algebra: A query may simply *reference* a relation $\underline{r}$ in the schema. *Renaming* allows giving a name to an intermediate query to be referenced later. Remember that $\underline{r}$ is an overloaded symbol that indicates both a relation and a relation name. A *projection* enables selecting a subset of attributes from the results of a subquery, for example, $\pi_{\underline{a}_1} \underline{r}$ would return only attribute $\underline{a}_1$ from $\underline{r}$. A *selection* enables filtering the tuples returned by a subquery based on a given condition $\underline{\theta}$, for example, $\sigma_{\underline{a}_1 > 3} \underline{r}$ would return all tuples from $\underline{r}$ where the value for $\underline{a}_1$ is greater than 3. The *join* operation joins two subqueries based on a condition and omitting its condition implies it is a natural join (i.e., join on the shared attribute of the two subqueries). For example, $\underline{r}_1 \bowtie_{\underline{a}_1 = \underline{a}_2} \underline{r}_2$ joins tuples from $\underline{r}_1$ and $\underline{r}_2$ where the attribute $\underline{a}_1$ from relation $\underline{r}_1$ is equal to attribute $\underline{a}_2$ from relation $\underline{r}_2$. However, if we have $\underline{r}_1(\underline{a}_1, \underline{a}_3)$ and $\underline{r}_2(\underline{a}_1, \underline{a}_2)$ then $\underline{r}_1 \bowtie \underline{r}_2$ joins tuples from $\underline{r}_1$ and $\underline{r}_2$ where attribute $\underline{a}_1$ has the same value in $\underline{r}_1$ and $\underline{r}_2$.

3.2 Formula Choice Calculus and Annotation

The choice calculus [27, 9] is a metalanguage for describing variation in programs and its elements such as data structures [28, 10]. In the choice calculus, variation is represented in-place as choices between alternative subexpressions. For example, the the variational expression $e = A\langle 1, 2\rangle + B\langle 3, 4\rangle + A\langle 5, 6\rangle$ contains three choices. Each choice has an associated *dimension*, which is used to synchronize the choice with other choices in different parts of the expression. For example, expression e contains two dimensions, A and B, and the two choices in dimension A are synchronized. Therefore, the variational expression e represents four different plain expressions, depending on whether the left or right alternatives are selected from each dimension. Assuming that dimensions evaluate to boolean values, we have: (1) $1+2+5$, A and B evaluate to `true`, (2) $1+4+5$, A evaluates to `true` and B evaluates to `false`, (3) $2+3+6$, A evaluates to `false` and B evaluates to `true`, and (4) $2+4+6$, A and B evaluate to `false`.

The formula choice calculus [14] extends the choice calculus by allowing dimensions to be propositional formulas. For example, the variational expression $e' = A \vee B\langle 1, 2\rangle$ represents two plain expressions: (1) 1, $A \vee B$ evaluate to `true` and (2) 1, $A \vee B$ evaluate to `false`.

Dimension e can also be used to *annotate/tag* element x which is denoted by x^e. For example, the variational expression $1^{A \vee B}$ states that the expression 1 is valid when $A \vee B$ evaluates to `true` and otherwise it is invalid.

Table 2: Objective 1 research questions.

Objective 1: Identify the kinds of variation existing in relational databases in different application domains
RQ1.1: What are the application domains that variation appears in a database? What are the dimensions of variation in a database? (Poly'18, VaMoS'21)
RQ1.2: How can all identified variation instances be represented in a generic encoding without regards for the application domain? (DBPL'17)
RQ1.3: Can we encode identified instances of variation using our encoding? What are the steps one need to take to encode an instance of variation in our encoding? (VaMoS'21)

4 Research Goals and Methods

4.1 Identify the kinds of variation existing in relational databases in different application domains

To encode variation explicitly in databases we investigate different kinds of variation that appears in databases in various application domains. Objective 1 aims to represent an encoding for variation that is generic enough that can encode different kinds of variation and is not bind to a specific instance of variation or application domain. Table 2 presents individual research questions we need to answer for this objective.

For RQ1.1 [complete with how variation arises in databases] we explored different application domains where databases change, yet, there is a need to keep and access the older versions (variants) of the database for business reasons.

For RQ1.2 we introduce a *feature space* that captures dimensions of variation that may appear in a variational database scenario. We then introduce *feature expressions* as propositional formulas of features. Section 4.1.1 explains this encoding in details.

For RQ1.3 ...

4.1.1 Encoding Variability

[fix examples to ones that you have introduced here]

To account for variability in a database we need a way to encode it. To encode variability we first organize the configuration space into a set of features, denoted by $\mathbf{F}$. For example, in the context of schema evolution, features can be generated from version numbers (e.g. features V_1 to V_5 and T_1 to T_5 in the motivating example, Table **??**); for SPLs, the features can be adopted from the SPL feature set (e.g. the *edu* feature in our motivating example, Table **??**); and for data integration, the features can be representatives of resources. For simplicity, the set of features is assumed to be closed and features are assumed to be boolean variables, however, it is easy to extend them to multi-valued variables that have finite set of values. A feature $f \in \mathbf{F}$ can be enabled (i.e., $f =$ `true`) or disabled ($f =$ `false`).

The features in $\mathbf{F}$ are used to indicate which parts of a variational entity within the database are different among different variants. Enabling or disabling each of the features in $\mathbf{F}$ produces a particular *variant* of the entity in which all variation has been removed. A *configuration* is a *total* function that maps every feature in the feature set to a boolean value. For brevity, we represent a configuration by the set of enabled features. For example, in our motivating scenario, the configuration $\{V_2, T_3, edu\}$ represents a database variant where

only features V_2, T_3, and *edu* are enabled. This database variant contains relation schemas of the employee and education sub-schemas associated with V_2 and T_3 in Table **??**, respectively. For brevity, we refer to a variant with configuration c as variant c. For example, variant $\{V_2, T_3, edu\}$ refers to the variant with configuration $\{V_2, T_3, edu\}$.

When describing variation points in the database, we need to refer to subsets of the configuration space. We achieve this by constructing propositional formulas of features. Thus, such a propositional formula defines a condition that holds for a subset of configurations and their corresponding variants. For example, the propositional formula $\neg edu$ represents all variants of our motivating example that do not have the education part of the schema, i.e., variant schemas of the left schema column.

We call a propositional formula of features a *feature expression* and define it formally in Figure 2. The evaluation function of feature expressions $\mathbb{E}[\![e]\!]_c : \mathbf{E} \rightarrow \mathbf{C} \rightarrow \mathbf{B}$ evaluates the feature expression e w.r.t. the configuration c. For example, $\mathbb{E}[\![f_1 \vee f_2]\!]_{\{f_1\}} = \texttt{true}$, however, $\mathbb{E}[\![f_1 \vee f_2]\!]_{\{\}} = \texttt{false}$, where the empty set indicates neither f_1 nor f_2 are enabled. Additionally, we define the binary *equivalence ($\equiv$)* relation and the unary *satisfiable (sat)* and *unsatisfiable (unsat)* relations over feature expressions in Figure 2.

To incorporate feature expressions into the database, we *annotate/tag* database elements (including attributes, relations, and tuples) with feature expressions. An *annotated element* x with feature expression e is denoted by x^e. The feature expression attached to an element is called a *presence condition* since it determines the condition (set of configurations) under which the element is present. $pc(x)$ returns the presence condition of the element x. For example, the annotated number $2^{f_1 \vee f_2}$ is present in variants with a configuration that enables either f_1 or f_2 or both but it does not exist in variants that disable both f_1 and f_2. Here, $pc(2) = f_1 \vee f_2$.

No matter the context, features often have a relationship with each other that constrains configurations. For example, only one of the temporal features of $V_1 - V_5$ can be `true` for a given variant. This relationship can be captured by a feature expression, called a *feature model* and denoted by m, which restricts the set of *valid configurations*: if configuration c violates the relationship then $\mathbb{E}[\![m]\!]_c$= `false`. For example, the restriction that at a given time only one of temporal features $V_1 - V_5$ can be enabled is represented by: $V_1 \oplus V_2 \oplus V_3 \oplus V_4 \oplus V_5$, where $f_1 \oplus f_2 \oplus \ldots \oplus f_n$ is syntactic sugar for $(f_1 \wedge \neg f_2 \wedge \ldots \wedge \neg f_n) \vee (\neg f_1 \wedge f_2 \wedge \ldots \wedge \neg f_n) \vee (\neg f_1 \wedge \neg f_2 \wedge \ldots \wedge f_n)$, i.e., features are mutually exclusive.

4.2 Design and implement a database framework that accommodate identified variations

Having an encoding that represent variation we need to incorporate it within the database and the query language to allow explicit storing and manipulation of variation in a database. Objective 2 aims to design and implement a database framework that considers variation as a first-class citizen. Table 3 presents individual research questions we need to answer for this objective.

For RQ2.1 ...

For RQ2.2 ...

For RQ2.3 ...

4.3 Demonstrate how the proposed system can be used to manage variation in databases in different application domains

Having a variational database framework, we need to examine how effectively it represents instances of variation in databases in different application domains. Objective 3 focuses on this goal and Table 4 represents individual research question we need to answer for this objective.

For RQ3.1 ...

Feature expression generic object:

$$f \in \mathbf{F} \quad \textit{Feature Name}$$

Feature expression syntax:

$$\begin{array}{lcll} b \in \mathbf{B} & ::= & \texttt{true} \mid \texttt{false} & \textit{Boolean Value} \\ e \in \mathbf{E} & ::= & b \mid f \mid \neg f \mid e \wedge e \mid e \vee e & \textit{Feature Expression} \\ c \in \mathbf{C} & = & \mathbf{F} \to \mathbf{B} & \textit{Configuration} \end{array}$$

Relations over feature expressions:

$$\begin{aligned} e_1 \equiv e_2 \textit{ iff } \forall c \in \mathbf{C} : \mathbb{E}[\![e_1]\!]_c = \mathbb{E}[\![e_2]\!]_c \\ \textit{sat}(e) \textit{ iff } \exists c \in \mathbf{C} : \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \textit{unsat}(e) \textit{ iff } \forall c \in \mathbf{C} : \mathbb{E}[\![e]\!]_c = \texttt{false} \end{aligned}$$

Figure 2: Feature expression syntax and relations.

Table 3: Objective 2 research questions.

Objective 2: Design and implement a database framework that accommodates identified variations
RQ2.1: How should variation in form of feature expression be incorporated in the database as a first-class citizen? (DBPL'17, Poly'18)
RQ2.2: What are appropriate query languages to interact with a database that accounts for variation explicitly? And how should variation in form of feature expression be incorporated in the query language? (DBPL'17, Poly'18)
RQ2.3: Having a theoretical database framework that accounts for variation explicitly, how should we go to implement a database management system that uses that framework? (In progress)

Table 4: Objective 3 research questions.

Objective 3: Demonstrate how the proposed system can be used to manage variation in databases in different application domains
RQ3.1: Can real-world instances of variation in databases be encoded as a VDB? What are the steps to generate a VDB from a scenario of variation in a database? (VaMoS'21)
RQ3.2: What are the benefits and drawbacks of representing variation generically in databases as opposed to having scenario-tailored approaches? (VaMoS'21)

Table 5: Objective 4 research questions.

Objective 4: Mechanize proofs of properties of the language and the system
RQ4.1: What are the desired properties for a VDB and do they hold for VDB? (VaMoS'21)
RQ4.2: What are the desired properties for VRA? Can they be mechanically proved? (In progress)
RQ4.3: Is the implementation of VDBMS compliant to semantics of VRA? (Not started)

For RQ3.2 ...

4.4 Mechanize proofs of properties of the language and the system

Having established that our framework effectively and explicitly encode variation within the database and its query language, we need to ensure that it satisfies the properties we desire. Objective 4 aims to define such properties and prove that they hold for our framework. Table 5 presents individual research questions we need to answer for this objective.

For RQ4.1 ...
For RQ4.2 ..
For RQ4.3 ...

4.5 Stretch goal: Generalize the encoding of variation to make the framework customizable for different application domains

The main goal of this research is to add variation as a first-class citizen to databases to separate the concern of dealing with variation from the application domain, however, as discussed in Section 4.3 there is a trade-off between expressiveness and complexity. In other words, although VDB allows one to encode different kinds of variation it introduces more complexity than a specialized system that only manages a specific instance of variation since it is not bind to the application domain. A possible workaround this problem is to generalize the encoding of variation to allow developers to customize variation to their use case such that it straps away unneeded complexity from the query language. We will explore this idea once objective 4 is completed and if time permits.

4.6 Summary

Figure 4.6 summarizes the connections between the research questions and activities. Table 4.6 provides the timeline for this proposal.

5 Related Work

The SPL community has a tradition of developing and distributing case studies to support research on software variation. For example, SPL2go [26] catalogs the source code and variability models of a large number of SPLs. Additionally, specific projects, such as Apel et al.'s [3] work on SPL verification, often distribute case studies along with study results. However, there are no existing datasets or case studies that include corresponding relational databases and queries, despite their ubiquity in modern software.

Table 6: Summary of projected and completed dates for each of the proposed research questions.

Research Question	Target conference	Projected Date	Status
1.1	Poly'18	N/A	Complete
1.2	DBPL'17	N/A	Complete
1.3	VaMoS'21	N/A	Complete
2.1	Poly'18, DBPL'17, VLDB'21	Early 2021	Complete
2.2	Poly'18, DBPL'17, VLDB'21	Early 2021	Complete
2.3	VLDB'21	Early 2021	In progress
3.1	VaMoS'21	N/A	Complete
3.2	VaMoS'21	N/A	Complete
4.1	VaMoS'21	N/A	Complete
4.2	VLDB'21, TOPLAS'21	Mid 2021	In progress
4.3	TOPLAS'21	Mid 2021	Not started

Many researchers have recognized the need to manage structural variation in the databases that SPLs rely on. Abo Zaid and De Troyer [2] argue for modeling data variability as part of a model-oriented SPL process. Their *variable data models* link features to concepts in a data model so that specialized data models can be generated for different products. Khedri and Khosravi [17] address data model variability in the context of delta-oriented programming. They define delta modules that can incrementally generate a relational database schema, and so can be used to generate different database schemas for each variant of a SPL. Humblet et al. [15] present a tool to manage variation in the schema of a relational database used by a SPL. Their tool enables linking features to elements of a schema, then generating different variants of the schema for different products. Schäler et al. [23] generates variable database schema from a given global schema and software variants configurations by mapping schema elements to features. Siegmund et al. [24] emphasizes the need for a variable database schema in SPL and proposes two decomposition approaches: (1) physical where database sub-schemas associated with a feature are stored in physical files and (2) virtual where a global Entity-Relation model of a schema is annotated with features. All of these approach address the issue of *structural* database variation in SPLs and provide a technique to derive a database schema per variant, which is also achievable by configuring a VDB. The work of Humblet et al. [15] is most similar to our notion of a variational schema since it is an annotative approach [16] that enables directly associating schema elements with features. Abo Zaid and De Troyer [2] is also annotative, but operates at the higher level of a data model that may only later be realized as a relational database. Khedri and Khosravi [17] is a compositional approach [16] to generating database schemas. None of these approaches consider *content-level* variation, which is captured by VDBs and observable in our case studies, nor do they consider how to express queries over databases with structural variation, which is addressed by our *variational queries*.

While the previous approaches all address data variation in space, Herrmann et al. [11] emphasizes that as a SPL evolves over time, so does its database. Their approach adapts work on database evolution to the domain of software product lines, enabling the safe evolution of all of the various products that have been deployed. They present the DAVE toolkit to address database evolution in SPL. Their approach generates a global evolution script from the local evolution scripts by grouping them into a single database operations and executing them sequentially. This approach requires having the old and new schema of a variant to generate the delta scripts. However, it uses these scripts to ensure correct evolution of both data and schema at the deployment step.

Database researchers have also studied several kinds of database variation in both time and space. There

is a substantial body of work on *schema evolution* and *database migration* [8, 21, 12, 22], which corresponds to variation in time. Typically the goal of such work is to safely migrate existing databases forward to new versions of the schema as it evolves. Work on *database versioning* [5, 13] shifts this idea to content level. In a versioned database, content changes can be sent between different instances of a database, similar to a distributed revision control system. All of this work is different from variational databases because it typically does not require maintaining or querying multiple versions of the database at once.

The representation of v-schemas and variational tables is based on previous work on variational sets [10], which is part of a larger effort toward developing safe and efficient variational data structures [28, 20]. The central motivation of work on variational data structures is that many applications can benefit from maintaining and computing with variation at runtime. Implementing SPL analyses are an example of such an application, but there are many more [28].

Variational research:

Schema evolution: Current solutions addressing schema evolution rely on temporal nature of schema evolution. They use timestamps as a means to keep track of historical changes either in an external document [21] or as versions attached to databases [19, 6, 4, 25], i.e., either approach fails to incorporate the timestamps into the database. Then, they take one of these approaches: 1) they require the DBA to design a unified schema, map all schema variants to the unified one, migrate the database variants to the unified schema, and write queries only on the unified schema [12], 2) they require the DBA to specify the version for their query and then migrating all database variants to the queried version [19, 6, 4, 25], or 3) they require the user to specify the timestamps for their query and then reformulate the query for other database variants [21]. These approaches satisfy **N0** by migration techniques. However, they cannot satisfy **N1** and **N2** because they only consider variation w.r.t. time and dismiss variation in space, resulting in querying all variants, i.e., variants cannot be queried selectively, and missing which variants data belongs to. Although these approaches do not provide a direct way to satisfy **N3** one can satisfy **N3** for them by manipulating the database with the schema evolution history file.

SPL and its evolution including its artifacts evolution: As mentioned in Section 1, SPLs use a database with universal schema for all variants of database, which is burdensome and time-consuming for SPL developers and DBAs [**?**]. While SPLs have techniques to satisfy **N0** and **N3** they do not have sufficient techniques to satisfy **N1** and **N2**. The problem of schema evolution when a SPL evolves is currently addressed by designing a new domain-specific language so that SPL developers can write scripts of the schema changes [11], which requires a great effort by DBAs and SPL developers. This approach only provides mechanisms for **N0** and **N3**. Note that the amount of work grows exponentially as the number of potential variants grows, a concerning behavior because a SPL usually has hundreds of features [18]. As the SPL and its database evolve, manually managing the variants becomes virtually impossible.

Database versioning: As mentioned in Section **??**, database versioning approaches only consider content-level variation [13] which is usually used for experimental and scientific databases.

6 Conclusion

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley, 1994.

[2] Lamia Abo Zaid and Olga De Troyer. Towards modeling data variability in software product lines. In Terry Halpin, Selmin Nurcan, John Krogstie, Pnina Soffer, Erik Proper, Rainer Schmidt, and Ilia Bider, editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 453–467, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-21759-3.

[3] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 482–491. IEEE, 2013.

[4] Gad Ariav. Temporally oriented data definitions: Managing schema evolution in temporally oriented databases. *Data & Knowledge Engineering*, 6(6):451 – 467, 1991. ISSN 0169-023X. doi: https://doi.org/10.1016/0169-023X(91)90023-Q. URL `http://www.sciencedirect.com/science/article/pii/0169023X9190023Q`.

[5] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow.*, 8 (12):1346–1357, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824035. URL `http://dx.doi.org/10.14778/2824032.2824035`.

[6] Cristina De Castro, Fabio Grandi, and Maria Rita Scalas. Schema versioning for multitemporal relational databases††recommended by peri loucopoulos. *Information Systems*, 22(5):249 – 290, 1997. ISSN 0306-4379. doi: https://doi.org/10.1016/S0306-4379(97)00017-3. URL `http://www.sciencedirect.com/science/article/pii/S0306437997000173`.

[7] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2001. ISBN 0-201-70332-7.

[8] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: The prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453939. URL `http://dx.doi.org/10.14778/1453856.1453939`.

[9] Martin Erwig and Eric Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011.

[10] Martin Erwig, Eric Walkingshaw, and Sheng Chen. An Abstract Representation of Variational Graphs. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2013.

[11] Kai Herrmann, Jan Reimann, Hannes Voigt, Birgit Demuth, Stefan Fromm, Robert Stelzmann, and Wolfgang Lehner. Database evolution for software product lines. In *DATA*, 2015.

[12] Jean-Marc Hick and Jean-Luc Hainaut. Database application evolution: A transformational approach. *Data & Knowledge Engineering*, 59(3):534 – 558, 2006. ISSN 0169-023X. doi: https://doi.org/10.1016/j.datak.2005.10.003. URL `http://www.sciencedirect.com/science/article/pii/S0169023X05001631`. Including: ER 2003.

[13] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya Parameswaran. Orpheusdb: Bolt-on versioning for relational databases. *Proc. VLDB Endow.*, 10(10):1130–1141, June 2017. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=3115404.3115417.

[14] Spencer Hubbard and Eric Walkingshaw. Formula Choice Calculus. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 49–57. ACM, 2016.

[15] Mathieu Humblet, Dang Vinh Tran, Jens H. Weber, and Anthony Cleve. Variability management in database applications. In *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*, VACE '16, pages 21–27, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4176-9. doi: 10.1145/2897045.2897050. URL http://doi.acm.org/10.1145/2897045.2897050.

[16] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 311–320, 2008.

[17] Niloofar Khedri and Ramtin Khosravi. Handling database schema variability in software product lines. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 331–338, 2013. doi: 10.1109/APSEC.2013.52. URL https://doi.org/10.1109/APSEC.2013.52.

[18] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. pages 105–114, 2010. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806819. URL http://doi.acm.org/10.1145/1806799.1806819.

[19] E. McKenzie and Richard Thomas Snodgrass. Schema evolution and the relational algebra. *Inf. Syst.*, 15(2):207–232, May 1990. ISSN 0306-4379. doi: 10.1016/0306-4379(90)90036-O. URL http://dx.doi.org/10.1016/0306-4379(90)90036-O.

[20] Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Int. Work. on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 28–35. ACM, 2017.

[21] Hyun J. Moon, Carlo A. Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and querying transaction-time databases under schema evolution. *Proc. VLDB Endow.*, 1(1):882–895, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453952. URL http://dx.doi.org/10.14778/1453856.1453952.

[22] Sudha Ram and Ganesan Shankaranarayanan. Research issues in database schema evolution: the road not taken. 2003.

[23] Martin Schäler, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Building information system variants with tailored database schemas using features. In Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanislaw Wrycza, editors, *Advanced Information Systems Engineering*, pages 597–612, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31095-9.

[24] Norbert Siegmund, Christian Kästner, Marko Rosenmüller, Florian Heidenreich, Sven Apel, and Gunter Saake. Bridging the gap between variability in client application and database schema. In

Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, and Gottfried Vossen, editors, *Datenbanksysteme in Business, Technologie und Web (BTW) - 13. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS)*, pages 297–306, Bonn, 2009. Gesellschaft f´ur Informatik e.V.

[25] Richard Thomas Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, USA, 1995. ISBN 0792396146.

[26] T Thüm and F Benduhn. SPL2go: An Online Repository for Open-Source Software Product Lines, 2011. `http://spl2go.cs.ovgu.de`.

[27] Eric Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, 2013. `http://hdl.handle.net/1957/40652`.

[28] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226, 2014.