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

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. However, there are instances of the general problem that are not well-studied, resulting in using manual approaches that burden experts. Moreover, different kinds of variation can interact which cannot be addressed by current approaches due to lack of a general solution to managing different kinds of variation in databases. In this paper, we provide a fundamental solution to managing variation in databases by considering variation explicitly as a *first-class citizen* in the system, allowing for encoding different kinds of variation.

Variation in databases arises when multiple database instances conceptually represent the same database, but, differ slightly either in their schema and/or content. The variation in schema and/or content occurs in two dimensions: time and space. Variation in space refers to different variants of database that coexist in parallel while variation in time refers to the evolution of database, similar to variation observed in software [**?**]. Note that variation in a database can occur due to both dimensions at the same time.

Schema evolution is an instance of schematic variation in databases that is well-supported [17, 5, 3, 23, 19]. 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 [17, 5, 3, 23] or keeping an external file of time-line history of changes applied to the database [19]. 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 [6]. 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.

Software evolution is unavoidable, so is its artifacts evolution, including databases [10]. This is where two instances of managing variation in databases (schema evolution and database-backed software developed by SPL) interact. While there are solutions to schema evolution they cannot adapt to a new situation because they only provide a solution to variation of databases in time and cannot encode the interaction of database variation in time and space. We motivate this case through an example in Section **??**.

As we have shown, variation in databases w.r.t. time and space is abundant and inexorable [24]; impacts DBAs, data scientists, and developers significantly [10]; and appears in different contexts. Various research has studied variation of databases in time and space, however, they all consider only one of these dimensions and are extremely tailored to a specific context. This becomes a problem when the two dimensions and different contexts interact. Hence, the challenge becomes incorporating variation in databases s.t. it can model different instances of variation in different contexts and it satisfies different specialists' needs at different stages, e.g., development, information extraction, and deployment. Our contributions in this paper address this challenge:

- We provide a framework to capture variation within a database using propositional formulas over sets of features, called *feature expressions*, following [**?**].
- We incorporate feature expressions into both the structure (schema) and content (tuples) of the database, introducing *variational schemas*, Section **??**, and *variational tables*, Section **??**, and together *VDBs*, Section **??**.
- To express user information needs we define the *variational relational algebra* query language, a combination of relational algebra and choice calculus [8, 26], Section **??**. Users query a VDB by a *variational query*, Section **??**.
- To make variational queries more useable and easier to understand, respectively, we define a static type

system, Section **??**, and *variation-minimiztion* rules, Section **??**.

- To query a variational database and receive clear results we implement *Variational Database Management System (VDBMS)*, Section **??**.

VDBMS provides new functionality to databases by accounting for schematic and content-level variation. We provide two use cases of how VDBMS can model the schema evolution and SPL instances of variation in databases for two real-world databases, Section **??**. We hypothesize that it especially pays off when there is lots of variation in the database and the user needs to query many of the variants simultaneously, Section **??**.

To answer these 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.

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 paper, 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

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 instance by *database*.

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

Name	Notation	Section
Feature	f	Section 3.2
Feature expression	e	
Annotated element x by e	x^e	
Configuration	c	
Evaluation of e under c	$\mathbb{E}[\![e]\!]_c$	
Presence condition of entity x	$pc(x)$	
Optional attribute	a	Section **??**
Variational attribute set	A	
Variational relation schema	s	
Variational schema	S	
Variational tuple	u	Section **??**
Variational relation content	U	
Variational table	$t = (s, U)$	
Choice	$e\langle x,y\rangle$	Section **??**
Variational condition	θ	
Variational query	q	

3.2 Encoding Variability

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 1. 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

Feature expression generic object:

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

Feature expression syntax:

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

Relations over feature expressions:

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

Figure 1: Feature expression syntax and relations.

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 1.

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.

3.3 Variational Set

A *variational set (v-set)* $X = \{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}$ is a set of annotated elements [9, 27, **?**]. Conceptually, a *variational set* represents many different plain sets that can be generated by enabling or disabling features and including only the elements whose feature expressions evaluate to `true`. We typically omit the presence condition `true` in a variational set, e.g., the v-set $\{2^{f_1}, 3^{f_2}, 4\}$ represents four plain sets under different configurations. These plain sets can be generated by *configuring* the variational set with a given configuration: $\{2,3,4\}$, when f_1 and f_2 are enabled; $\{2,4\}$, when f_1 is enabled but f_2 is disabled; $\{3,4\}$, when f_2 is enabled but f_1 is disabled; and $\{4\}$, when both f_1 and f_2 are disabled. Note that elements with presence condition `false` can be omitted from the v-set, e.g., the v-set $\{1^{\texttt{false}}\}$ is equivalent to an empty v-set. For simplicity and to comply with database notational conventions we drop the brackets of a variational set when used in database schema definitions and queries.

A variational set itself can also be annotated with a feature expression. $X^e = \{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}^e$ is an *annotated v-set*. Annotating a v-set with the feature expression e restricts the condition under which its elements are present, i.e., it forces elements' presence conditions to be more specific. This restriction can be applied to all elements of the set by *pushing* in the feature expression e, done by the operation $\downarrow(\{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}^e) = \{{x_1}^{e_1 \wedge e}, \ldots, {x_n}^{e_n \wedge e}\}$. For example, the annotated v-set $\{2^{f_1}, 3^{\neg f_2}, 4, 5^{f_3}\}^{f_1 \wedge f_2}$ indicates that all the elements of the set can only exist when both f_1 and f_2 are enabled. Thus, pushing in the set's feature expression results in $\{2^{f_1 \wedge f_2}, 4^{f_1 \wedge f_2}, 5^{f_1 \wedge f_2 \wedge f_3}\}$. The element 3 is dropped since $\neg sat(\neg f_2 \wedge (f_1 \wedge f_2))$, where $pc(3) = \neg f_2 \wedge (f_1 \wedge f_2)$.

We provide some operations over v-sets. Intuitively, these operations should behave such that configuring the result of applying a variational set operation should be equivalent to applying the plain set operation on the configured input v-sets.

Definition 1 (V-set union)**.** *The* union *of two v-sets is the union of their elements with the disjunction of presence conditions if an element exists in both v-sets:* $X_1 \cup X_2 = \{x^{e_1} \mid x^{e_1} \in X_1, x^{e_2} \notin X_2\} \cup \{x^{e_2} \mid x^{e_2} \in X_2, x^{e_1} \notin X_1\} \cup \{x^{e_1 \vee e_2} \mid x^{e_1} \in X_1, x^{e_2} \in X_2\}$*. For example,*
$\{2, 3^{e_1}, 4^{e_1}\} \cup \{3^{e_2}, 4^{\neg e_1}\} = \{2, 3^{e_1 \vee e_2}, 4\}$.

Definition 2 (V-set intersection)**.** *The* intersection *of two v-sets is a v-set of their shared elements annotated with the conjunction of their presence conditions, i.e.,* $X_1 \cap X_2 = \{x^{e_1 \wedge e_2} \mid x^{e_1} \in X_1, x^{e_2} \in X_2, sat(e_1 \wedge e_2)\}$*. For example,* $\{2, 3^{f_1}, 4^{\neg f_2}\} \cap \downarrow\left(\{2, 3, 4, 5\}^{f_2}\right) = \{2^{f_2}, 3^{f_1 \wedge f_2}\}$.

Definition 3 (V-set cross product)**.** *The* cross product *of two v-sets is a pair of every two elements of them annotated with the conjunction of their presence conditions.* $X_1 \times X_2 = \{(x_1, x_2)^{e_1 \wedge e_2} \mid x_1^{e_1} \in X_1, x_2^{e_2} \in X_2\}$

Definition 4 (V-set equivalence)**.** *Two v-sets are* equivalent*, denoted by* $X_1 \equiv X_2$*, iff* $\forall x^e \in (X_1 \cup X_2).x^{e_1} \in X_1, x^{e_2} \in X_2, e_1 \equiv e_2$*, i.e., they both cover the same set of elements and the presence conditions of elements from the two v-sets are equivalent.*

Definition 5 (V-set subsumption)**.** *The v-set* X_1 subsumes *the v-set* X_2*,* $X_2 \prec X_1$*, iff* $\forall x^{e_2} \in X_2.x^{e_1} \in X_1, sat(e_2 \wedge e_1)$*, i.e., all elements in* X_2 *also exist in* X_1 *s.t. the element is valid in a shared configuration between the v-sets. For example,* $\downarrow\left(\{2, 3\}^{f_1}\right) \prec \{2, 3^{f_1 \vee f_2}, 4\}$*, however,* $\downarrow\left(\{2, 3\}^{f_1}\right) \not\prec \{2, 3^{\neg f_1 \wedge f_2}\}$ *and* $\{2^{f_1}, 3^{f_1}, 4\} \not\prec \{2, 3^{f_1 \wedge f_2}\}$.

4 Research Goals and Methods

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

4.2 Design a query language and implement a database management system that accommodate identified variations

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

4.4 Mechanize proofs of properties of the language and the system

4.5 Summary

Figure **??** summarizes the connections between the research questions and activities. Table **??** 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 [25] catalogs the source code and variability models of a large number of SPLs. Additionally, specific projects, such as Apel et al.'s [2] 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.

Many researchers have recognized the need to manage structural variation in the databases that SPLs rely on. [1] 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. [15] 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. [13] 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. [21] generates variable database schema from a given global schema and software variants configurations by mapping schema elements to features. [22] 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 [13] is most similar to our notion of a variational schema since it is an annotative approach [14] that enables directly associating schema elements with features. [1] is also annotative, but operates at the higher level of a data model that may only later be realized as a relational database. [15] is a compositional approach [14] 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, [10] 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* [7, 19, 11, 20], 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* [4, 12] 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 [9], which is part of a larger effort toward developing safe and efficient variational data structures [27, 18]. 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 [27].

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 [19] or as versions attached to databases [17, 5, 3, 23], 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 [11], 2) they require the DBA to specify the version for their query and then migrating all database variants to the queried version [17, 5, 3, 23], or 3) they require the user to specify the timestamps for their query and then reformulate the query for other database variants [19]. 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 [10], 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 [16]. 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 [12] which is usually used for experimental and scientific databases.

6 Conclusion

References

[1] Abo Zaid, L., and De Troyer, O. Towards modeling data variability in software product lines. In *Enterprise, Business-Process and Information Systems Modeling* (Berlin, Heidelberg, 2011), T. Halpin, S. Nurcan, J. Krogstie, P. Soffer, E. Proper, R. Schmidt, and I. Bider, Eds., Springer Berlin Heidelberg, pp. 453–467.

[2] Apel, S., von Rhein, A., Wendler, P., Grösslinger, A., and Beyer, D. Strategies for product-line verification: case studies and experiments. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 482–491.

[3] Ariav, G. Temporally oriented data definitions: Managing schema evolution in temporally oriented databases. *Data & Knowledge Engineering 6*, 6 (1991), 451 – 467.

[4] Bhattacherjee, S., Chavan, A., Huang, S., Deshpande, A., and Parameswaran, A. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1346–1357.

[5] Castro, C. D., Grandi, F., and Scalas, M. R. Schema versioning for multitemporal relational databases††recommended by peri loucopoulos. *Information Systems 22*, 5 (1997), 249 – 290.

[6] Clements, P., and Northrop, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2001.

[7] Curino, C. A., Moon, H. J., and Zaniolo, C. Graceful database schema evolution: The prism workbench. *Proc. VLDB Endow. 1*, 1 (Aug. 2008), 761–772.

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

[9] Erwig, M., Walkingshaw, E., and Chen, S. An Abstract Representation of Variational Graphs. In *Int. Work. on Feature-Oriented Software Development (FOSD)* (2013), ACM, pp. 25–32.

[10] Herrmann, K., Reimann, J., Voigt, H., Demuth, B., Fromm, S., Stelzmann, R., and Lehner, W. Database evolution for software product lines. In *DATA* (2015).

[11] Hick, J.-M., and Hainaut, J.-L. Database application evolution: A transformational approach. *Data & Knowledge Engineering 59*, 3 (2006), 534 – 558. Including: ER 2003.

[12] Huang, S., Xu, L., Liu, J., Elmore, A. J., and Parameswaran, A. Orpheusdb: Bolt-on versioning for relational databases. *Proc. VLDB Endow. 10*, 10 (June 2017), 1130–1141.

[13] Humblet, M., Tran, D. V., Weber, J. H., and Cleve, A. Variability management in database applications. In *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design* (New York, NY, USA, 2016), VACE '16, ACM, pp. 21–27.

[14] Kästner, C., Apel, S., and Kuhlemann, M. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering* (2008), pp. 311–320.

[15] Khedri, N., and Khosravi, R. Handling database schema variability in software product lines. In *Asia-Pacific Software Engineering Conference (APSEC)* (2013), pp. 331–338.

[16] LIEBIG, J., APEL, S., LENGAUER, C., KÄSTNER, C., AND SCHULZE, M. An analysis of the variability in forty preprocessor-based software product lines. pp. 105–114.

[17] MCKENZIE, E., AND SNODGRASS, R. T. Schema evolution and the relational algebra. *Inf. Syst. 15*, 2 (May 1990), 207–232.

[18] MENG, M., MEINICKE, J., WONG, C.-P., WALKINGSHAW, E., AND KÄSTNER, C. A Choice of Variational Stacks: Exploring Variational Data Structures. In *Int. Work. on Variability Modelling of Software-Intensive Systems (VaMoS)* (2017), ACM, pp. 28–35.

[19] MOON, H. J., CURINO, C. A., DEUTSCH, A., HOU, C.-Y., AND ZANIOLO, C. Managing and querying transaction-time databases under schema evolution. *Proc. VLDB Endow. 1*, 1 (Aug. 2008), 882–895.

[20] RAM, S., AND SHANKARANARAYANAN, G. Research issues in database schema evolution: the road not taken.

[21] SCHÄLER, M., LEICH, T., ROSENMÜLLER, M., AND SAAKE, G. Building information system variants with tailored database schemas using features. In *Advanced Information Systems Engineering* (Berlin, Heidelberg, 2012), J. Ralyté, X. Franch, S. Brinkkemper, and S. Wrycza, Eds., Springer Berlin Heidelberg, pp. 597–612.

[22] SIEGMUND, N., KÄSTNER, C., ROSENMÜLLER, M., HEIDENREICH, F., APEL, S., AND SAAKE, G. Bridging the gap between variability in client application and database schema. In *Datenbanksysteme in Business, Technologie und Web (BTW) - 13. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS)* (Bonn, 2009), J.-C. Freytag, T. Ruf, W. Lehner, and G. Vossen, Eds., Gesellschaft f́ur Informatik e.V., pp. 297–306.

[23] SNODGRASS, R. T. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, USA, 1995.

[24] STONEBRAKER, M., DENG, D., AND BRODIE, M. L. Database decay and how to avoid it. In *Big Data (Big Data), 2016 IEEE International Conference* (2016), IEEE.

[25] THÜM, T., AND BENDUHN, F. SPL2go: An Online Repository for Open-Source Software Product Lines, 2011. `http://spl2go.cs.ovgu.de`.

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

[27] WALKINGSHAW, E., KÄSTNER, C., ERWIG, M., APEL, S., AND BODDEN, E. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software (Onward!)* (2014), pp. 213–226.