Should Variation Be Encoded Explicitly in Databases?

Parisa Ataei
ataeip@oregonstate.edu
Oregon State University
Corvallis, Oregon, USA

Qiaoran Li
liqiao@oregonstate.edu
Oregon State University
Corvallis, Oregon, USA

Eric Walkingshaw
walkiner@oregonstate.edu
Oregon State University
Corvallis, Oregon, USA

ABSTRACT

Variation occurs in databases in many different forms and contexts. For example, a single database schema evolves over time, data from different sources may be combined, and the various configurations of a software product line (SPL) may have different data needs. While approaches have been developed to deal with many such scenarios, particularly in the fields of database evolution and data integration, there is no solution that treats variation as a general and orthogonal concern in databases. This is a problem when various kinds of variation intersect, such as during the evolution of a SPL. Previously, we have proposed variational databases (VDB) as a general way to represent variation in both the structure and content of databases. Although the model underlying VDB is simple, encoding variation explicitly in databases introduces complexity akin to using preprocessing directives in software. In this paper, we explore the feasibility of VDB and its associated variational query language for encoding different kinds of database variability. We develop two use cases that illustrate how different kinds of variation can be encoded and integrated in VDB, and how the corresponding information needs can be expressed as variational queries. We then use these use cases to discuss the benefits and drawbacks of such a direct encoding of variation in data and queries.

KEYWORDS

Variational databases, variational queries, variability in data

ACM Reference Format:
Parisa Ataei, Qiaoran Li, and Eric Walkingshaw. 2021. Should Variation Be Encoded Explicitly in Databases?. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21), February 9–11, 2021, Krems, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3442391.3442395

1 INTRODUCTION

Just as variation is ubiquitous in software, it is also ubiquitous in the relational databases that software systems rely on. Database researchers have long studied different kinds of variation in databases, such as through work on database evolution [12, 36, 38], database versioning [10, 28], and data integration [18]. However, work in the databases community does not identify *variation* as a general, orthogonal concern that arises in many different contexts. This is a problem since it means that tools and techniques developed for one kind of database variation cannot be easily reused for another. More concretely for software developers, and especially software product line (SPL) practitioners, the lack of a general representation of database variation means that many kinds of variation that arise in software do not map cleanly onto corresponding variation encodings in the databases they use.

In contrast, SPL researchers have invested significant effort in studying *variation* (or *variability*) as a general phenomenon and concern in software. Although many kinds of software variation are possible, most can be roughly organized into variation in *time* or *space* [47]. Variation in *time* refers to the evolution of a system and is addressed by revision control systems and configuration management [17], while variation in *space* refers to the simultaneous development and maintenance of related systems with different feature sets and is the traditional focus of research on SPLs [4]. Multiple lines of work in the SPL community have sought to develop general purpose representations of variation in software, such as delta-oriented programming [40] and the choice calculus [19], among others. These can be used to unify both kinds of variation in software, enabling reuse of analyses across dimensions and enabling new kinds of analyses that consider variation in both time and space simultaneously [47].

Variation in databases can also be organized into *time* and *space* dimensions. Variation in time occurs as a database evolves while variation in space occurs as different data needs arise due to different variants of a software system or different information sources. However, work on different dimensions of variation in databases have remained much more separate than in SPLs. For example, consider work on schema evolution, which addresses the fact that database schemas change over time as business requirements evolve. This problem has been studied extensively by the databases community [6, 12, 36, 38]. However, the solutions rely fundamentally on the temporal nature of this kind of variation, using timestamps or linear histories of changes. While addressing the schema evolution problem as formulated, these solutions are unsuitable for encoding other kinds of variation in schemas, such as occurs when developing a database-backed SPL [42]. Similarly, work on database versioning [10, 28], data integration [18], and data provenance [11] are all closely tied to the particular kinds of data variation they address, without addressing the general phenomenon.

Because the existing data variation models developed in the databases community do not align with all of the variation scenarios that arise in software, SPL researchers have identified the need for more general encodings of variation in data models and database schema. To this end, they have developed encodings of data models that allow for arbitrary variation by annotating different elements of the model with features from the SPL [2, 41, 42].

VaMoS'21, February 9–11, 2021, Krems, Austria

ACM ISBN 978-1-4503-8824-5/21/02…$15.00
https://doi.org/10.1145/3442391.3442395

However, these solutions address only variation in the data model but do not extend to the level of the data or queries. The lack of variation support in queries leads to unsafe techniques such as encoding different variants of query through string munging, while the lack of variation support in data precludes testing with multiple variants of a database at once.

In previous work, we have developed *variational databases (VDBs)* and *variational queries* [7, 8] to encode general variation in the representation and use of relational databases. Our work extends ideas developed in the SPL community to relational databases. Conceptually, a VDB represents potentially many different plain relational databases at the same time. Similarly, a variational query represents potentially many different queries, each one corresponding to a variant of the VDB. Together, VDBs and variational queries enable safely and efficiently working with many variants of a relational database at once, and reliably integrating the variants of a database with the corresponding variants of an SPL. We are currently implementing these ideas in *VDBMS*, a practical implementation of variational databases as a lightweight wrapper on top of a traditional relational database management system.

However, the generic and expressive approach of VDB in dealing with database variation creates new complexity and costs which raises the question: Is explicitly encoding variation in databases actually a good idea? With this question in mind, in this paper, we:

- Show the feasibility of VDB by systematically generating two VDBs from realistic scenarios of database variation in time and space (Section 3).
- Illustrate the applicability of variational queries by encoding information needs for the developed VDBs using scenarios described in the literature (Section 4).
- Discuss the tradeoffs of explicitly encoding variation in databases (Section 5).

We distribute the VDBs, SQL scripts for generating them, and queries of our use cases.[1] We distribute the VDBs in both MySQL and Postgres in two forms, one intended for use with our VDBMS tool, and one intended for more general-purpose research on variation in databases. We distribute the variational queries as simple `#ifdef`-annotated SQL files to promote their broad reuse in the design and evaluation of other systems for managing variational relational data.

2 BACKGROUND

In this section, we provide background and notation needed in the rest of the paper. A *variational database (VDB)* conceptually represents multiple database variants that represent the same database, but, slightly differ in their structure (i.e. schema) and/or content. A *variational query (v-query)* conceptually represents multiple queries written over different database variants.

The variation space of a VDB is organized into boolean variables called *features*, similar to many SPLs. A propositional formula of features is called a *feature expression*. Feature expressions are used to *annotate* elements of the database and queries. Such annotations are called *presence conditions* because they determine the conditions under which each element is present, that is, which configurations of the VDB or variational query contain those elements. A *configuration* is a mapping from features to boolean values and is used to *configure* a VDB into a plain relational database variant.

The structure of a VDB is defined by a *variational schema (v-schema)* and its content consists of a set of *variational tables*. A variational schema is a set of relation schemas, where each relation and each attribute in each relation is also annotated by a presence condition. The overall variational schema is also annotated by a presence condition called a *feature model*, denoted by m. A variational table corresponds to a relation in the schema and is a set of annotated tuples. Due to the hierarchical structure of a VDB, m implicitly annotates all of the relations in the schema, and each relation's annotation implicitly annotates each of its attributes and each of the tuples in its corresponding table. Since not all attributes of a relation are present in all configurations, we use `NULL` values in tuples for attributes that are not present in any configurations that the tuple is present. Example 2.1 illustrates these concepts.

Example 2.1. Assume a VDB has two features f_1 and f_2, one relation r, and a trivial feature model $m = \texttt{true}$ (i.e. all configurations of f_1 and f_2 are valid). The relation schema of r is $r(a_1, a_2{}^{\neg f_2})^{f_1}$, where annotated elements are indicated by superscripts. The outermost f_1 annotation indicates that r is present only when f_1 is enabled, while the annotation $\neg f_2$ attached to attribute a_2 indicates that attribute a_2 is present only when f_2 is disabled (and when its relation r is present). The attribute a_1 is unannotated, which indicates a presence condition of `true`, which we omit for brevity. We use $pc(\cdot)$ to refer to the presence condition attached to a given element within a VDB. For example, the fully-expanded presence condition of a_2, taking the hierarchy of the variational schema into account, is $m \wedge pc(r) \wedge pc(a_2) = \texttt{true} \wedge f_1 \wedge \neg f_2$.

The content of the VDB consists of a single table corresponding to relation r: $\{(1, 2), (3, \texttt{NULL})^{f_2}\}$. The second element of the second tuple is `NULL` since the tuple is only present when f_2 is enabled but the corresponding attribute a_2 is present only when f_2 is disabled. There are two non-empty configurations of this VDB: (1) For configuration $f_1 = \texttt{true}, f_2 = \texttt{true}$, the resulting plain database has the schema $\{r(a_1)\}$ and content $\{(1), (3)\}$. (2) For configuration $f_1 = \texttt{true}, f_2 = \texttt{false}$, the resulting plain database has the schema $\{r(a_1, a_2)\}$ and content $\{(1, 2)\}$. The other two possible configurations of the VDB, where $f_1 = \texttt{false}$, contain no relations.

To query a VDB, we need a query language that explicitly accounts for variation. We use variational relational algebra (VRA) for this purpose, whose syntax is defined in Figure 1. VRA is a combination of relational algebra [1] with the formula choice calculus [29], which is a formal language for representing variation. The first five constructs are adapted from relational algebra: A query may simply *reference* a relation r in the schema. *Renaming* allows giving a name to an intermediate query to be referenced later. A *projection* enables selecting a subset of attributes from the results of a subquery, for example, $\pi_{a_1} r$ would return only attribute a_1 from r; we extend the standard project operator to work with annotated lists of attributes, for example, $\pi_{a_1, a_2^e} r$ would include a_1 for all configurations and also a_2 for configurations where e is true. A *selection* enables filter the tuples returned by a subquery based on a given condition θ, for example, $\sigma_{a_1 > 3} r$ would return all tuples from r where the value

[1] Available at: https://zenodo.org/record/4321921

$$
\begin{array}{rcll}
e \in \mathbf{E} & ::= & \texttt{true} \mid \texttt{false} \mid f \mid \neg f \mid e \wedge e \mid e \vee e & \\
\theta \in \mathbf{\Theta} & ::= & \texttt{true} \mid \texttt{false} \mid a \bullet k \mid a \bullet a \mid \neg\theta \mid \theta \vee \theta & \\
 & \mid & \theta \wedge \theta \mid e\langle\theta, \theta\rangle & \\
q \in \mathbf{Q} & ::= & r & \textit{Relation reference} \\
 & \mid & \rho_r q & \textit{Renaming} \\
 & \mid & \pi_A q & \textit{Projection} \\
 & \mid & \sigma_\theta q & \textit{Selection} \\
 & \mid & q \bowtie_\theta q & \textit{Join} \\
 & \mid & e\langle q, q\rangle & \textit{Choice} \\
 & \mid & \varepsilon & \textit{Empty relation}
\end{array}
$$

Figure 1: Syntax of variational relational algebra, where $\bullet$ ranges over comparison operators ($<, \leq, =, \neq, >, \geq$), k over cosntant values, a over attribute names, and A over lists of annotated attributes. The syntactic category e represents feature expressions, θ is variational conditions, and q is variational relational algebra terms.

for a_1 is greater than 3; these conditions may be variational to enable returning different tuples for different configurations of the VDB. The *join* operation is the standard relational join operation and omitting its condition implies it is a natural join (i.e. a join on the shared attribute of the two subqueries). A *choice* encodes a variation point between two subquery alternatives based on a given feature expression, e.g., $f_1 \wedge f_2\langle q_l, q_r\rangle$ yields the results of q_l alternative for configurations where f_1 and f_2 are enabled, and in other configurations yields the results of q_r alternative. Note that the conditions θ used by selections and joins also contain choices, and these behave similarly. Finally, the *empty relation* returns a trivial empty query. It is mostly used as an alternative in choices when a query is only relevant in some configurations.

A v-query can be configured into a set of plain relational queries. The unique non-empty plain relational queries that a v-query can be configured to are called its *variants*. For example, the query $\pi_{a_1, a_2 \neg f_2} r$ over the VDB given in Example 2.1 has two variants: $\pi_{a_1} r$ for configuration $f_1 = \texttt{true}$, $f_2 = \texttt{true}$ and $\pi_{a_1, a_2} r$ for configuration $f_1 = \texttt{true}$, $f_2 = \texttt{false}$.

3 VARIATIONAL DATABASES

In this section, we show how to encode the variation of two realistic database variation scenarios as VDBs. The use cases presented in Section 3.1 and Section 3.2 illustrate the effects of variation in space and time, respectively. For each use case, we show how all variants of the database can be captured in a single VDB. Finally, Section 3.3 discusses properties of well-formed VDBs that we use to validate our use cases.

3.1 Variation in Space: Email SPL Use Case

Our first use case focuses on variation in space. It shows the use of VDB to encode the variational information needs of a database-backed SPL. We consider an email SPL that has been used in several previous SPL research projects (e.g. [3, 5]). Our use case is formed by systematically combining two pre-existing works: (1) We use Hall's decomposition of an email system into its component features [25] as high-level specification of a SPL. (2) We use the Enron email dataset[2] as a realistic email database. In combining these works, we show how variation in space in an email SPL requires corresponding variation in a supporting database, how we can link the variation in the software to variation in the database, and how all of these variants can be encoded in a single VDB.

Table 1: Original Enron email dataset schema.

employeelist(*eid*, *firstname*, *lastname*, *email_id*, *email2*, *email3*, *email4*, *folder*, *status*)
messages(*mid*, *sender*, *date*, *message_id*, *subject*, *body*, *folder*)
recipientinfo(*rid*, *mid*, *rtype*, *rvalue*)
referenceinfo(*rid*, *mid*, *reference*)

3.1.1 Variation Scenario: An Email SPL. The email SPL consists of the following features from Hall [25]: *addressbook*, users can maintain lists of known email addresses with corresponding aliases, which may be used in place of recipient addresses; *signature*, messages may be digitally signed and verified using cryptographic keys; *encryption*, messages may be encrypted before sending and decrypted upon receipt using cryptographic keys; *autoresponder*, users can enable automatically generated email responses to incoming messages; *forwardmessages*, users can forward all incoming messages automatically to another address; *remailmessage*, users may send messages anonymously; *filtermessages*, incoming messages can be filtered according to a provided white list of known sender address suffixes; and *mailhost*, a list of known users is maintained and known users may retrieve messages on demand while messages sent to unknown users are rejected.

Hall's decomposition separates *signature* and *encryption* into two features each (corresponding to signing and verifying, encrypting and decrypting). Since these pairs of features must always be enabled together, we reduce them to one feature each for simplicity.

The listed features are used in presence conditions within the v-schema for the email VDB, linking the software variation to variation in the database. In the email SPL, each feature is optional and independent, resulting in the simple feature model $m_{en} = \texttt{true}$.

3.1.2 Generating V-Schema of the Email SPL VDB. To produce a v-schema for the email VDB, we start from plain schema of the Enron email dataset shown in Table 1, then systematically adjust its schema to align with the information needs of the email SPL described by Hall [25]. The *employeelist* table contains information about the employees of the company. The *messages* table contains information about the email messages. The *recipientinfo* table contains information about the recipient of a message. The *referenceinfo* table contains messages that have been referenced in other email messages. This table simply backs up the emails.

From this starting point, we introduce new attributes and relations that are needed to implement the features in the email SPL. We attach presence conditions to new attributes and relations corresponding to the features they are needed to support, which ensure they will *not* be present in configurations that do not include the relevant features. The resulting v-schema is given in Table 2.

For example, consider the *signature* feature. In the software, implementing this feature requires new operations for signing an email before sending it out and for verifying the signature of a received email. These new operations suggest new information

[2] http://www.ahschulz.de/enron-email-data/

Table 2: V-schema of the email VDB with feature model m_{en}. Presence conditions are colored blue for clarity.

employeelist(*eid*, *firstname*, *lastname*, *email_id*, *folder*, *status*, *verification_key*^{signature}, *public_key*^{encryption})	*recipientinfo*(*rid*, *mid*, *rtype*, *rvalue*)
messages(*mid*, *sender*, *date*, *message_id*, *subject*, *body*, *folder*, *is_system_notification*, *is_encrypted*^{encryption}, *is_autoresponse*^{autoresponder}, *is_signed*^{signature}, *is_forward_msg*^{forwardmessages})	*forward_msg*(*eid*, *forwardaddr*)^{forwardmessages}
	mailhost(*eid*, *username*, *mailhost*)^{mailhost}
filter_msg(*eid*, *suffix*)^{filtermessages} *remail_msg*(*eid*, *pseudonym*)^{remailmessage} *auto_msg*(*eid*, *subject*, *body*)^{autoresponder} *alias*(*eid*, *email*, *nickname*)^{addressbook}	

needs: we need a way to indicate that a message has been signed, and we need access to each user's public key to verify those signatures (private keys used to sign a message would not be stored in the database). These needs are reflected in the v-schema by the new attributes *verification_key* and *is_signed*, added to the relations *employeelist* and *messages*, respectively. The new attributes are annotated by the *signature* presence condition, indicating that they correspond to the *signature* feature and are unused in configurations that exclude this feature. Additionally, several features require adding entirely new relations, e.g., when the *forward_msg* feature is enabled, the system must keep track of which users have forwarding enabled and the address to forward the messages to. This need is reflected by the new *forward_msg* relation, which is correspondingly annotated by the *forward_msg* presence condition.

A main focus of Hall's decomposition [25] is on the many feature interactions. Several of the features may interact in undesirable ways if special precautions are not taken. For example, any combination of the *forward_msg*, *remail_msg*, and *autoresponder* features can trigger an infinite messaging loop if users configure the features in the wrong way; preventing this creates an information need to identify auto-generated emails, which is realized in the variational schema by attributes like *is_forward_msg* and *is_autoresponse*.

For brevity, we omit some attributes and relations from the original schema that are irrelevant to the email SPL described by Hall.

We provide the v-schema both in the encoding used by our VDBMS tool and also in plain SQL. The SQL encoding is given by a "universal" schema containing the relations and attributes of all variants, plus a relation *vdb_pcs*(*element_id*, *pres_cond*) that captures all of the relevant presence conditions. The plain SQL encoding of the v-schema supports the use of the use cases for research on the effective management of variation in databases independent of VDBMS.

3.2 Variation in Time: Employee Use Case

Our second use case focuses on variation in time by demonstrating the use of a VDB to encode an employee database evolution scenario systematically adapted from Moon et al. [38] and populated by a dataset that is widely used in databases research.[3]

3.2.1 Variation Scenario: An Evolving Employee Database. Moon et al. [38] describe an evolution scenario in which the schema of a company's employee management system changes over time, yielding the five versions of the schema shown in Table 3. In V_1, employees are split into two separate relations for engineer and non-engineer personnel. In V_2, these two tables are merged into one relation, *empacct*. In V_3, departments are factored out of the *empacct* relation and into a new *dept* relation to reduce redundancy in the database. In V_4, the company decides to start collecting more personal information about their employees and stores all personal

[3] https://github.com/datacharmer/test_db

Table 3: Evolution of an employee database schema [38].

Version	Schema
V_1	*engineerpersonnel* (*empno*, *name*, *hiredate*, *title*, *deptname*) *otherpersonnel* (*empno*, *name*, *hiredate*, *title*, *deptname*) *job* (*title*, *salary*)
V_2	*empacct* (*empno*, *name*, *hiredate*, *title*, *deptname*) *job* (*title*, *salary*)
V_3	*empacct* (*empno*, *name*, *hiredate*, *title*, *deptno*) *job* (*title*, *salary*) *dept* (*deptname*, *deptno*, *managerno*)
V_4	*empacct* (*empno*, *hiredate*, *title*, *deptno*) *job* (*title*, *salary*) *dept* (*deptname*, *deptno*, *managerno*) *empbio* (*empno*, *sex*, *birthdate*, *name*)
V_5	*empacct* (*empno*, *hiredate*, *title*, *deptno*, *salary*) *dept* (*deptname*, *deptno*, *managerno*) *empbio* (*empno*, *sex*, *birthdate*, *firstname*, *lastname*)

Table 4: Employee v-schema with feature model m_{emp}.

engineerpersonnel(*empno*, *name*, *hiredate*, *title*, *deptname*)^{V_1}
otherpersonnel(*empno*, *name*, *hiredate*, *title*, *deptname*)^{V_1}
empacct(*empno*, *name*$^{V_2 \vee V_3}$, *hiredate*, *title*, *deptname*^{V_2}, *deptno*$^{V_3 \vee V_4 \vee V_5}$, *salary*^{V_5})$^{V_2 \vee V_3 \vee V_4 \vee V_5}$
job(*title*, *salary*)$^{V_2 \vee V_3 \vee V_4}$
dept(*deptname*, *deptno*, *managerno*)$^{V_3 \vee V_4 \vee V_5}$
empbio(*empno*, *sex*, *birthdate*, *name*^{V_4}, *firstname*^{V_5}, *lastname*^{V_5})$^{V_4 \vee V_5}$

information in the new relation *empbio*. Finally, in V_5, the company decides to decouple salaries from job titles and instead base salaries on individual employee's qualifications and performance; this leads to dropping the *job* relation and adding a new *salary* attribute to the *empacct* relation. This version also separates the *name* attribute in *empbio* into *firstname* and *lastname* attributes.

We associate a feature with each version of the schema, named $V_1 \ldots V_5$. These features are mutually exclusive since only one version of the schema is valid at a time. This yields the feature model:

$$m_{emp} = (V_1 \wedge \neg V_2 \wedge \neg V_3 \wedge \neg V_4 \wedge \neg V_5) \vee (\neg V_1 \wedge V_2 \wedge \neg V_3 \wedge \neg V_4 \wedge \neg V_5) \vee (\neg V_1 \wedge \neg V_2 \wedge V_3 \wedge \neg V_4 \wedge \neg V_5) \vee (\neg V_1 \wedge \neg V_2 \wedge \neg V_3 \wedge V_4 \wedge \neg V_5) \vee (\neg V_1 \wedge \neg V_2 \wedge \neg V_3 \wedge \neg V_4 \wedge V_5)$$

3.2.2 Generating V-Schema of the Employee VDB. The v-schema for this scenario is given in Table 4. It encodes all five of the schema versions in Table 3 and was systematically generated by the following process. First, generate a universal schema from all of the plain schema versions; the universal schema contains every relation and attribute appearing in any of the five versions. Then, annotate the attributes and relations in the universal schema according to the versions they are present in. For example, the *empacct* relation is present in versions V_2–V_5, so it will be annotated by the feature

expression $V_2 \vee V_3 \vee V_4 \vee V_5$, while the *salary* attribute within the *empacct* relation is present only in version V_5, so it will be annotated by simply V_5. Since the presence conditions of attributes are implicitly conjuncted with the presence condition of their relation, we can avoid redundant annotations when an attribute is present in all instances of its parent relation. For example, the *empbio* relation is present in $V_4 \vee V_5$, and the *birthdate* attribute is present in the same versions, so we do not need to redundantly annotate it.

3.3 Properties of Well-Formed VDBs

In this section, we describe a set of basic properties that a well-formed VDB should satisfy. These checks ensure that presence conditions are consistent and satisfiable, which ensures that each element is present in at least one variant. In the following, $sat(e)$ denotes a satisfiability check that returns *true* if the feature expression e is satisfiable and *false* otherwise.

A well-formed v-schema should have the following properties:

(1) There is at least one valid configuration of the feature model m: $sat(m)$
(2) Every relation r is present in at least one configuration of the variational schema: $\forall r \in S, sat(m \wedge pc(r))$
(3) Every attribute a in every relation r is present in at least one configuration of the variational schema: $\forall a \in r, \forall r \in S, sat(m \wedge pc(r) \wedge pc(a))$
(4) If S_c denotes the expected plain relational schema for configuration c of the variational schema S, then configuring the variational schema with that configuration, written $[\![S]\!]_c$, actually yields that variant: $\forall c \in \mathbf{C}, [\![S]\!]_c = S_c$

At the data level, a well-formed VDB should have these properties:

(1) Every tuple u in relation r is present in at least one variant: $\forall u \in r, \forall r \in S, sat(m \wedge pc(r) \wedge pc(u))$
(2) For every tuple u in relation r, if an attribute a in r is not present in any variants of the tuple, then the value of that attribute in the tuple, written $value_u(a)$, should be NULL: $\forall u \in r, \forall a \in r, \forall r \in S, \neg sat(m \wedge pc(r) \wedge pc(a) \wedge pc(u)) \Rightarrow value_u(a) = \text{NULL}$

We implemented these checks in our VDBMS tool and verified that both use cases described in this paper satisfy all of them. Depending on the context of the VDB, more specialized properties can be checked too. For example, if temporal variability in a database is accumulated over variants (i.e. old data is included in more recent variants in addition to newly added data), it is desirable to ensure that older variants are subsets of newer variants. This property should hold for our employee data set. To check this, assume that configurations $c_1, c_2, \cdots$ represent time-ordered configurations, then check $\forall c_i, c_j \in \mathbf{C}, i \leq j, [\![D]\!]_{c_i} \subseteq [\![D]\!]_{c_j}$, where $[\![D]\!]_c$ denotes configuring the VDB instance D for configuration c.

4 VARIATIONAL QUERIES

Variation in software affects not only databases but also how developers and database administrators interact with databases. Since different software variants have different information needs, developers must often write and maintain different queries for different software variants. Moreover, even if a particular information need is similar across variants, different variants of a query may need to be created and maintained to account for structural differences in the schema for each variant. Creating and maintaining different queries for each variant is tedious and error-prone, and potentially even intractable for large and open-ended configuration spaces, such as most open-source projects [43].

In this section, we illustrate how variation in software leads to variation in information needs and the queries that realize those information needs. We also show how variational information needs can be captured by *v-queries* written in VRA. For each use case, we provide a set of v-queries and we present a sample of them here.

We distribute the v-queries in two formats: (1) VRA, encoded in the format used by our VDBMS tool, and (2) plain SQL queries with embedded `#ifdef` annotations to capture variation points.[4] The SQL format provides queries for studying variational data independently of VDBMS tool, but we use VRA in this paper for its brevity.

4.1 Email Query Set

To produce a set of queries for the email SPL use case, we collected all of the information needs that we could identify in the description of the email SPL by Hall [25]. In order to make the information needs more concrete, we viewed the requirements of the email SPL mostly through the lens of constructing an email header. An email header includes all of the relevant information needed to send an email and is used by email systems and clients to ensure that an email is sent to the right place and interpreted correctly. Although there is obviously other infrastructure involved, the fundamental information needs of an email system can be understood by considering how to construct email headers.

Hall's decomposition focuses on enumerating the features of the email SPL and enumerating the potential interactions of those features. We deduce the information need for each feature by asking: "what information is needed to modify the email header in a way that incorporates the new functionality?". We deduce the information need for each interaction by asking: "what information is needed to modify the email header in a way that avoids the undesirable feature interaction?". We can then translate these information needs into queries on the underlying variational database.

In total, we provide 27 queries for the email SPL. This consists of 1 query for constructing the basic email header, 8 queries for realizing the information needs corresponding to each feature, and 18 queries for realizing the information needs to correctly handle the feature interactions described by Hall.

We start by presenting the query to assemble the basic email header, Q_{basic}. This corresponds to the information need of a system with no features enabled. We use X to stand for the specific message ID (*mid*) of the email whose header we want to construct.

$$Q_{basic} = \pi_{sender, rvalue, subject, body} mes\_rec$$
$$mes\_rec \leftarrow (\sigma_{mid=X} messages) \bowtie recipientinfo$$

Taking Q_{basic} as our starting point, we next construct our set of 8 *single-feature queries* that capture the information needs specific to each feature. When a feature is enabled in the SPL, more information is needed to construct the header of email X. For example, if the feature *filtermessages* is enabled, then the query Q_{filter} extends Q_{basic} with the *suffix* attribute used in filtering. This additional

[4] Complete sets of queries in both formats are available at: https://zenodo.org/record/4321921

information allows the system to filter a message if its address contains any of the suffixes set by the receiver.

$$Q_{filter} = \pi_{sender, rvalue, suffix, subject, body}\, temp$$
$$temp \leftarrow mes\_rec\_emp \bowtie filter\_msg$$
$$mes\_rec\_emp \leftarrow mes\_rec \bowtie_{rvalue=email\_id} employeelist$$

We can construct a query that retrieves the required header information whether *filtermessages* is enabled or not by combining Q_{basic} and Q_{filter} in a choice, as $Q_{bf} = filtermessages\langle Q_{filter}, Q_{basic}\rangle$. Although we do not show the process in this paper, we can use equivalence laws from the choice calculus [19, 29] to factor commonalities out of choices and reduce redundancy in queries like Q_{bf}. The other single-feature queries are written similarly.

Besides single-feature queries, we also provide queries that gather information needed to identify and address the undesirable feature interactions described by Hall [25]. Out of Hall's 27 feature interactions, we determined 16 of them to have corresponding information needs related to the database; 2 of the interactions require 2 separate queries to resolve. Therefore, we define and provide 18 queries addressing all 16 of the relevant feature interactions. As before, we deduced the information needs through the lens of constructing an email header; in these cases, the header would correspond to an email produced after successfully resolving the interaction. However, some interactions can only be detected but not automatically resolved. In these cases, we constructed a query that would retrieve the relevant information to detect and report the issue.

One undesirable feature interaction occurs between the *signature* and *forwardmessages* features: if Philippe signs a message and sends it to Sarah, and Sarah forwards the message to an alternate address Sarah-2, then signature verification may incorrectly interpret Sarah as the sender rather than Philippe and fail to verify the message (Hall's interaction #4). A solution to this interaction is to embed the original sender's verification information into the email header of the forwarded message so that it can be used to verify the message, rather than relying solely on the message's "from" field.

Below, we show a variational query Q_{sf} that includes four variants corresponding to whether *signature* and *forwardmessages* are enabled or not independently. The information need for resolving the interaction is satisfied by the first alternative of the outermost choice with condition $signature \wedge forwardmessages$. The alternatives of the choices nested to the right satisfy the information needs for when only *signature* is enabled, only *forwardmessages* is enabled, or neither is enabled (Q_{basic}). We don't show the single-feature Q_{sig} query, but it is similar to other single-feature queries shown above.

$$\begin{aligned} Q_{sf} = {} & signature \wedge forwardmessages \\ & \langle \pi_{rvalue, forwardaddr, emp1.is\_signed, emp1.verification\_key}\, temp, \\ & signature\langle Q_{sig}, forwardmessages\langle Q_{forward}, Q_{basic}\rangle\rangle\rangle \\ temp \leftarrow {} & ((((\sigma_{mid=X}\, messages) \bowtie recipientinfo) \\ & \bowtie_{sender=emp1.email\_id} (\rho_{emp1} employeelist)) \\ & \bowtie_{rvalue=emp2.email\_id} (\rho_{emp2} employeelist)) \bowtie forward\_msg \end{aligned}$$

Some feature interactions require more than one query to satisfy their information need. For example, assume both *encryption* and *forwardmessages* are enabled. Philippe sends an encrypted email X to Sarah; upon receiving it the message is decrypted and forwarded it to Sarah-2 (Hall's interaction #9). This violates the intention of encrypting the message and the system should warn the user. Queries Q_{ef} and Q'_{ef} satisfy the information need for this interaction when a message is encrypted or unencrypted, respectively.

$$\begin{aligned} Q_{ef} = {} & encryption \wedge forwardmessages \\ & \langle \pi_{rvalue}(\sigma_{mid=X \wedge is\_encrypted}\, messages), encryption\langle Q_{encrypt}, \\ & forwardmessages\langle Q_{forward}, Q_{basic}\rangle\rangle \\ Q'_{ef} = {} & encryption \wedge forwardmessages\langle temp, encryption\langle Q_{encrypt}, \\ & forwardmessages\langle Q_{forward}, Q_{basic}\rangle\rangle\rangle \\ temp \leftarrow {} & \pi_{rvalue, forwardaddr, subject, body}(\sigma_{mid=X \wedge \neg is\_encrypted} \\ & (mes\_rec\_emp \bowtie_{employeelist.eid=forward\_msg.eid} forward\_msg)) \end{aligned}$$

However, managing feature interactions is not necessarily complicated. Some interactions simply require projecting more attributes from the corresponding single-feature queries. For example, assume both *filtermessages* and *mailhost* features are enabled. Philippe sends a message to a non-existant user in a mailhost that he has filtered. The mailhost generates a non-delivery notification and sends it to Philippe, but he never receives it since it is filtered out (Hall's interaction #26). The system can check the *is_system_notification* attribute for the Q_{filter} query and decide whether to filter a message or not. Therefore, we can resolve this interaction by extending the single-feature query for *filtermessages* to Q'_{filter}.

$$Q'_{filter} = \pi_{sender, rvalue, suffix, is\_system\_notification, subject, body}\, temp$$
$$temp \leftarrow mes\_rec\_emp \bowtie_{employeelist.eid=filter\_msg.eid} filter\_msg$$

Overall, for the 18 interaction queries we provide, 12 have 4 variants, 3 have 3 variants, 2 have 2 variants, and 1 has 1 variant.

4.2 Employee Query Set

For this use case, we have a set of existing plain queries to start from. Moon et al. [38] provides 12 queries to evaluate the Prima schema evolution system. We adapt these queries to fit our encoding of the employee VDB described in Section 3.2. 9 of these queries have one variant, 2 have two variants, and 1 has three variants.

Moon's queries are of two types: 6 retrieve data valid on a particular date (corresponding to V_3 in our encoding), while 6 retrieve data valid on or after that date (V_3–V_5 in our encoding). For example, one query expresses the intent "return the salary of employee number 10004" at a time corresponding to V_3, which we encode:

$$Q_1 = \pi_{salary^{V_3}}\, (\sigma_{empno=10004}\, empacct) \bowtie_{empacct.title=job.title} job.$$

We encode the same intent, but for all times at or after V_3 as follows:

$$\begin{aligned} Q_2 = {} & V_3 \vee V_4 \vee V_5\langle \pi_{salary}(V_3 \vee V_4\langle \\ & ((\sigma_{empno=10004}\, empacct)) \bowtie job, \sigma_{empno=10004}\, empacct\rangle), \varepsilon\rangle \end{aligned}$$

There are a variety of ways we could have encoded both Q_1 and Q_2. For Q_1 we could equivalently have embedded the projection in a choice, $V_3\langle \pi_{salary}(\ldots), \varepsilon\rangle$, however attaching the presence condition to the only projected attribute determines the presence condition of the resulting table and so achieves the same effect. In Q_2 we use choices to structure the query since we have to project on a different intermediate result for V_5 than for V_3 and V_4.

As another example, the following query realizes the intent to "return the name of the manager of department d001" during the time frame of V_3–V_5: $Q_3 = V_3 \vee V_4 \vee V_5\langle \pi_{name, firstname, lastname}$ $(V_3\langle empacct, empbio\rangle \bowtie_{empno=managerno} (\sigma_{deptno="d001"} dept)), \varepsilon\rangle$.

Note that even though the attributes *name*, *firstname*, and *lastname* are not present in all three of the variants corresponding to V_3–V_5, the VRA encoding permits omitting presence conditions that can be completely determined by the presence conditions of the corresponding relations or attributes in the variational schema. So, Q_3 is equivalent to the following query in which the presence conditions of the attributes from the variational schema are listed explicitly in the projection: $Q_3' = V_3 \vee V_4 \vee V_5\langle\pi_{name^{V_3 \vee V_4}, firstname^{V_5}, lastname^{V_5}}$ $(V_3\langle empacct, empbio\rangle \bowtie_{empno=managerno} (\sigma_{deptno="d001"} dept)), \varepsilon\rangle$. Allowing developers to encode variation in v-queries based on their preference makes VRA more flexible and easy to use. Also, v-queries are statically type-checked to ensure that the variation encoded in them does not conflict the variation encoded in the v-schema.

5 DISCUSSION

In this section we discuss the use cases and our encodings of VDB and v-queries in the context of the question posed in the title of this paper: *Should variation be encoded explicitly in databases?*

Expressiveness of explicit variation. The use cases in Section 3 and Section 4 show that by treating variation as an orthogonal concern and embedding it directly in databases and queries (via presence conditions and choices), one can encode data variation scenarios in both time and space. In fact, VDBs and v-queries are *maximally expressive* in the sense that any set of plain relational databases can be encoded as a single VDB and any set of plain queries over the variants of a VDB can be encoded as a v-query.[5]

The expressiveness of our approach is its main advantage over other ways to manage database variation. When working with a form of variation that already has its own specialized solution (e.g. schema evolution, data integration), the expressiveness of explicit variation is probably not worth the additional complexity. The expressiveness of explicit variation is most useful when working with a form of variation that is not well supported (e.g. query-level variation in SPLs), or when combining multiple forms of variation in one database (e.g. during SPL evolution).

We expect that ill-supported forms of variation are common in industry and justify the expressiveness of explicit variation. For example, the following is a scenario we recently discussed with an industry contact: A software company develops software for different networking companies and analyzes data from its clients to advise them accordingly. The company records information from each of its clients' networks in databases customized to the particular hardware, operating systems, etc. that each client uses. The company analysts need to query information from all clients who agreed to share their information, but the same information need will be represented differently for each client. This problem is essentially a combination of the SPL variation problem (the company develops and maintains many databases that vary in structure and content) and the data integration problem (querying over many databases that vary in structure and content). However, neither the existing solutions from the SPL community nor database integration address both sides of the problem. Currently the company manually maintains variant schemas and queries, but this does not take advantage of sharing and is a major maintenance challenge. With a database encoding that supports explicit variation in schemas, content, and queries, the company could maintain a single variational database that can be configured for each client, import shared data into a VDB, and write v-queries over the VDB to analyze the data, significantly reducing redundancy across clients.

Complexity of explicit variation. The generality of explicit variation comes at the cost of increased complexity. The complexity introduced by presence conditions and choices is similar to the complexity introduced by variation annotations in annotative approaches to SPL implementation [32]. There is widespread acknowledgment that unrestricted use of variation annotations, such as the C Preprocessor's #ifdef notation [24], makes software difficult to understand [34] and is error prone [23]. However, so-called *disciplined* use of variation annotations, where annotations are used in a way that is consistent with the object language syntax of variants, may suffer less from such issues [35]. In VDBs, and in the VRA notation for v-queries, annotations are disciplined since presence conditions and choices are integrated into the existing syntax of relational database schemas and relational algebra. Note that annotation discipline is not enforced in the #ifdef-annotated SQL notation that we use to distribute the v-queries associated with our use cases.

Subjectively, the development of our use cases suggests that the impact of variation annotations on understandability is moderate for v-schemas and VDBs, and significant for v-queries written in VRA, despite the fact that such annotations are disciplined.

It is possible that a more restrictive and/or coarse-grained form of variation in v-queries would make them easier to understand at the cost of increased redundancy and (potentially) reduced expressiveness. This tradeoff is one we already made when considering how to encode variation in the *content* of a VDB. Specifically, we do not support cell-level variation in a VDB (e.g. choices within individual cells). This does not reduce the expressiveness of content variation in VDBs since cell-level variation can be simulated by row variation, but it does increase redundancy since all non-varied cells in the row must be duplicated. Similarly, variation in queries could be restricted to expression-level choices, with no choices or annotations in conditions or attribute lists. This would likely make understanding individual query variants easier at the cost of increasing redundancy among the alternatives of each choice.

Alternatively, the understandability of v-queries could be improved through tooling, for example, using background colors [22], virtual separation of concerns [31], or view-based editing [44, 49]. Future work should validate our subjective assessment of the understandability VDBs and v-queries, and explore techniques for improving this concern.

Analyzability of explicit variation. The relationship of our work to alternative approaches can be viewed through the lens of annotative vs. compositional variation, familiar to the SPL community [32]. VDBs and v-queries rely on generic annotations embedded directly in schemas and queries, respectively, while approaches from the databases community often express variation through separate

[5] The expressiveness of VDBs and v-queries can be proved by construction. For VDBs, one can simply take the union of all relations, attributes, and tuples across all variants, then attach presence conditions corresponding to which variants each is present in. For v-queries, all variants can be organized under a tree of choices that similarly organizes the variants in the appropriate way.

artifacts, such as views [9]. Annotative vs. compositional representations often exhibit the same tradeoff between expressiveness and complexity described above: annotative variation tends to be general and expressive, while compositional variation tends to be more restrictive but support modular reasoning [32]. Traditionally, another advantage of compositional approaches is that they are more analyzable thanks to the ability to analyze components separately (i.e. *feature-based* analysis [45]), a benefit shared by database views. However, in the last decade there has been a significant amount of work in the SPL community to improve the analyzability of annotative variation by analyzing whole variational artifacts directly (i.e. *family-based* analysis [45]). Although not presented here, we build directly on this body of work, especially work on variational typing [13, 14], to enable efficiently checking v-queries against all variants of a VDB, among other properties. Thus, the increased complexity of explicit variation annotations does not prevent us from verifying its correctness.

6 RELATED WORK

We proposed encoding variation explicitly in database schemas and queries in [7] and proposed applying this idea to database-backed SPLs in [8]. Our previous work uses slightly different encodings; the one presented here is the basis of our VDBMS implementation. This is the first work that provides use cases for VDBs.

The SPL community has a tradition of developing and distributing use cases to support research on software variation. For example, SPL2go [46] catalogs the source code and variability models of a large number of SPLs. Additionally, specific projects, such as Apel et al.'s [5] work on SPL verification, often distribute use cases along with study results. However, there are no existing datasets or use cases 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. 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 [33] 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 schemas for each variant of a SPL. Humblet et al. [30] 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. [41] generate a variable database schema from a given global schema and software configurations by mapping schema element to features. Siegmund et al. [42] emphasize the need for variable database schema in SPLs and propose 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 approaches address the issue of *structural* database variation in SPLs and provide a way to derive a schema per variant, which is also achievable by configuring a VDB. The work of Humblet et al. [30] is most similar to our notion of a variational schema since it is an annotative approach [32] that directly associates 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 [33] is a compositional approach [32] to generating database schemas. None of these approaches consider *content-level* variation, which is captured by VDBs and observable in our use cases, nor do they consider how to express queries over databases with structural variation, which is addressed by our *v-queries*.

While the previous approaches all address data variation in space, Herrmann et al. [26] emphasize that as an SPL evolves over time, so does its database. Their approach adapts work on database evolution to SPLs, enabling the safe evolution of all deployed products.

Database researchers have studied several kinds of variation in both time and space. There is a substantial body of work on *schema evolution* and *database migration* [16, 27, 38, 39], 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* [10, 28] extends this idea to a database's content. 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 encodes a less general notion of variation and does not support querying multiple versions of the database at once. Work on *data integration* can be viewed as managing variation in space [18]. In data integration, the goal is to combine data from disparate sources and provide a unified interface for querying. This is different from VDBs, which make differences between variants explicit.

The representation of v-schemas and variational tables is based on previous work on variational sets [21], which is part of a larger effort toward developing safe and efficient variational data structures [37, 48]. The central motivation of work on variational data structures is that many applications can benefit from maintaining and computing with variation at runtime [15, 20]. The ability to maintain and query several variants of a database at once extends the idea of computing with variation to relational databases.

7 CONCLUSION

We provide two use cases that illustrate how software variation leads to corresponding variation in relational databases. These use cases demonstrate the feasibility of VDBs and v-queries to capture the data needs of variational software systems. We argue that effectively managing such variation is an open problem, and we believe that these use cases will form a useful basis for evaluating research that addresses it, such as our own VDBMS framework.

VDBs encode variation explicitly in the structure and content of databases. This is a source of complexity that may impact understandability, as can be observed in our use cases. However, it also has several advantages: it is general in the sense that any set of variant databases and queries can be encoded as a VDB and v-queries, and it enables directly associating variation in databases to variation in software. By applying variational typing to variational queries, this generality does not come at the cost of safety. Future work can explore how tooling can mitigate the usability concerns using techniques that have been developed in the SPL community.

REFERENCES

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

[2] Lamia Abo Zaid and Olga De Troyer. 2011. Towards Modeling Data Variability in Software Product Lines. In *Enterprise, Business-Process and Information Systems Modeling*, Terry Halpin, Selmin Nurcan, John Krogstie, Pnina Soffer, Erik Proper, Rainer Schmidt, and Ilia Bider (Eds.). Springer, Berlin, Heidelberg, 453–467.

[3] Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. 2019. Effective Product-Line Testing Using Similarity-Based Product Prioritization. *Software & Systems Modeling* 18, 1 (2019), 499–521.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-Oriented Software Product Lines.* Springer-Verlag, Berlin.

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

[6] Gad Ariav. 1991. Temporally oriented data definitions: Managing schema evolution in temporally oriented databases. *Data & Knowledge Engineering* 6, 6 (1991), 451 – 467. https://doi.org/10.1016/0169-023X(91)90023-Q

[7] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. 2017. Variational Databases. In *Int. Symp. on Database Programming Languages (DBPL)*. ACM, 11:1–11:4.

[8] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. 2018. Managing Structurally Heterogeneous Databases in Software Product Lines. In *VLDB Workshop: Polystores and Other Systems for Heterogeneous Data (Poly)*.

[9] François Bancilhon and Nicolas Spyratos. 1981. Update Semantics of Relational Views. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 557–575.

[10] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1346–1357. https://doi.org/10.14778/2824032.2824035

[11] Peter Buneman and Wang-Chiew Tan. 2007. Provenance in Databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (Beijing, China) *(SIGMOD '07)*. Association for Computing Machinery, New York, NY, USA, 1171?1173. https://doi.org/10.1145/1247480.1247646

[12] Cristina De Castro, Fabio Grandi, and Maria Rita Scalas. 1997. Schema Versioning for Multitemporal Relational Databases. *Information Systems* 22, 5 (1997), 249 – 290. https://doi.org/10.1016/S0306-4379(97)00017-3

[13] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2012. An Error-Tolerant Type System for Variational Lambda Calculus. In *ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*. 29–40.

[14] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems (TOPLAS)* 36, 1 (2014), 1:1–1:54.

[15] Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2016. A Calculus for Variational Programming. In *European Conf. on Object-Oriented Programming (ECOOP) (LIPIcs)*, Vol. 56. 6:1–6:26.

[16] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. 2008. Graceful Database Schema Evolution: The PRISM Workbench. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 761–772. https://doi.org/10.14778/1453856.1453939

[17] Susan Dart. 1991. Concepts in Configuration Management Systems. In *Int. Work. on Software Configuration Management*. 1–18.

[18] AnHai Doan, Alon Halevy, and Zachary Ives. 2012. *Principles of Data Integration* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

[20] Martin Erwig and Eric Walkingshaw. 2013. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering IV (GTTSE 2011), Revised and Extended Papers (LNCS)*, Vol. 7680. 55–99.

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

[22] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. 2013. Do Background Colors Improve Program Comprehension in the #ifdef Hell? *Empirical Software Engineering* 18, 4 (2013), 699–745.

[23] Gabriel Ferreira, Momin Malik, Christian Kästner, Juergen Pfeffer, and Sven Apel. 2016. Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel. In *Int. Software Product Line Conf.*

[24] GNU Project. 2009. *The C Preprocessor.* Free Software Foundation. http://gcc.gnu.org/onlinedocs/cpp/.

[25] Robert J. Hall. 2005. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering* 12, 1 (2005), 41âĂŞ79.

[26] Kai Herrmann, Jan Reimann, Hannes Voigt, Birgit Demuth, Stefan Fromm, Robert Stelzmann, and Wolfgang Lehner. 2015. Database Evolution for Software Product Lines. In *DATA*.

[27] Jean-Marc Hick and Jean-Luc Hainaut. 2006. Database application evolution: A transformational approach. *Data & Knowledge Engineering* 59, 3 (2006), 534 – 558. https://doi.org/10.1016/j.datak.2005.10.003 Including: ER 2003.

[28] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya Parameswaran. 2017. OrpheusDB: Bolt-on Versioning for Relational Databases. *Proc. VLDB Endow.* 10, 10 (June 2017), 1130–1141. http://dl.acm.org/citation.cfm?id=3115404.3115417

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

[30] Mathieu Humblet, Dang Vinh Tran, Jens H. Weber, and Anthony Cleve. 2016. Variability Management in Database Applications. In *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design* (Austin, Texas) *(VACE '16)*. ACM, New York, NY, USA, 21–27. https://doi.org/10.1145/2897045.2897050

[31] Christian Kästner and Sven Apel. 2009. Virtual Separation of Concerns—A Second Chance for Preprocessors. *Journal of Object Technology* 8, 6 (2009), 59–78.

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

[33] Niloofar Khedri and Ramtin Khosravi. 2013. Handling Database Schema Variability in Software Product Lines. In *Asia-Pacific Software Engineering Conference (APSEC)*. 331–338. https://doi.org/10.1109/APSEC.2013.52

[34] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing (VL/HCC)*. 143–150.

[35] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Int. Conf. on Aspect-Oriented Software Development*. 191–202.

[36] Edwin McKenzie and Richard Thomas Snodgrass. 1990. Schema Evolution and the Relational Algebra. *Inf. Syst.* 15, 2 (May 1990), 207–232. https://doi.org/10.1016/0306-4379(90)90036-O

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

[38] Hyun J. Moon, Carlo A. Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. 2008. Managing and Querying Transaction-time Databases Under Schema Evolution. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 882–895. https://doi.org/10.14778/1453856.1453952

[39] Sudha Ram and Ganesan Shankaranarayanan. 2003. Research Issues in Database Schema Evolution: the Road Not Taken.

[40] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Int. Conf. on Software Product Lines*. Springer, 77–91.

[41] Martin Schäler, Thomas Leich, Marko Rosenmüller, and Gunter Saake. 2012. Building Information System Variants with Tailored Database Schemas Using Features. In *Advanced Information Systems Engineering*, Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanislaw Wrycza (Eds.). Springer, Berlin, Heidelberg, 597–612.

[42] Norbert Siegmund, Christian Kästner, Marko Rosenmüller, Florian Heidenreich, Sven Apel, and Gunter Saake. 2009. Bridging the Gap Between Variability in Client Application and Database Schema. In *13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*. Gesellschaft für Informatik (GI), 297–306.

[43] Micheal Stonebraker, Dong Deng, and Micheal L. Brodie. 2016. Database Decay and How to Avoid It. In *Big Data (Big Data), 2016 IEEE International Conference*. IEEE. https://doi.org/10.1109/BigData.2016.7840584

[44] Ştefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*. 323–333.

[45] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)* 47, 1 (2014), 6.

[46] Thomas Thüm and Fabian Benduhn. 2011. SPL2go: An Online Repository for Open-Source Software Product Lines. http://spl2go.cs.ovgu.de.

[47] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Toward Efficient Analysis of Variation in Time and Space. In *Int. Work. on Variability and Evolution of Software Intensive Systems (VariVolution)*.

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

[49] Eric Walkingshaw and Klaus Ostermann. 2014. Projectional Editing of Variational Software. In *ACM SIGPLAN Int. Conf. on Generative Programming: Concepts and Experiences (GPCE)*. 29–38.