Theory and Implementation of a Variational Database Management System

Parisa S. Ataei
Thesis Proposal
Submitted November 10th, 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* as an orthogonal concern of relational databases. We extend relational database theory to incorporate variation explicitly in databases and queries: we define *variational schemas* for describing variation in the structure of a database, *variational databases* for capturing variation in content, and *variational queries* for expressing variation in information needs with an accompanying type system that statically type checks the queries. Although the model underlying variational database is simple, encoding variation explicitly in databases introduces complexity akin to using preprocessing directives in software. We evaluate the feasibility of this approach by systematically developing two case studies that illustrate how different kinds of variation needs can be encoded and integrated in a variational database and how the corresponding information needs can be expressed as variational queries. We also design and implement a variational database management system as an abstraction layer over a traditional relational database. We demonstrate the applicability and feasibility of our approach on our two case-studies.

1 Introduction

Variation in databases arises when multiple database instances conceptually represent the same database, but, differ slightly either in their schema and/or content. Managing variation in databases is a perennial problem in database literature and appears in different forms and contexts. Specific cases of this problem has been extensively studied including schema evolution [33, 15, 6, 35], data integration [21], and database versioning [11, 27], where each instance has a context-specific solution that is tailored to the constrained problem definition. For example, schema evolution approaches deal with variation in a schema (i.e., the structure of the data) over time, and database versioning and data integration systems manage variation in the content of the database. Yet, the database community does not consider all of these as instances of a similar problem.

Consider schema evolution which is an instance of schematic variation in databases that is well-supported [33, 15, 6, 42, 35]. 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 the temporal nature of schema evolution by using timestamps [33, 15, 6, 42] or keeping an external history of changes applied to the database [35]. 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 context-specific systems that break when used in another instance of variation in databases.

Additionally, the software product line (SPL) community has realized that variation in software propagates to the artifacts the software relies on, including the database. [revise def. Eric said it's awkward!] 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 [18]. Different products of an SPL typically have different sets of enabled features or are tailored to run in different environments. These differences impose different data requirements, which in turn requires a different database for each product. Researchers have addressed this problem by encoding variation explicitly in the data model, called a *variable data model*, by associating features with elements of the schema. The specific database for a client's application can then be generated by only selecting the set of elements associated with features that have been selected for the client's application [41, 39, 2]. However, this approach breaks when software itself evolves over time, which is unavoidable [25].

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

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

- Objective 1: Identify the kinds of variation existing in relational databases in different application domains.
- Objective 2: Design a query language and implement a database management system that accommodate variations identified in Objective 1.
- Objective 3: Demonstrate how the proposed system manages variation in databases in different application domains and how effective and efficient it is.
- Objective 4: Mechanize proofs of properties of the language and the system.

[at the end, make sure the abstract and objectives are promising the same thing]

Each objective has some research questions that need to be answered which are followed by how we approached or will approach questions with references to sections including details of our approach.

2 Statement of Thesis

The goal of this research is to provide the theory and implementation fo variational databases. The theory should define the database and the query language, including its type system and semantics. The implementation should be aligned with the semantics of the query language.

3 Preliminaries

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

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

Name	Notation	Section
Feature	f	Section 4.1.2
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 4.2.2
Variational attribute set	A	
Variational relation schema	s	
Variational schema	S	
Variational tuple	u	Section 4.2.3
Variational relation content	U	
Variational table	$t = (s, U)$	
Choice	$e\langle x, y\rangle$	Section 4.2.4
Variational condition	θ	
Variational query	q	

3.1 Relational Databases and Relational Algebra

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

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

Figure 1 defines the syntax of relational algebra which allows users to query a relational database [1]. The first five constructs are adapted from relational algebra: A query may simply *reference* a relation $\underline{r}$ in the schema. *Renaming* allows giving a name to an intermediate query to be referenced later. Note that $\underline{r}$ is an overloaded symbol that indicates both a relation and a relation name. A *projection* enables selecting a subset of attributes from the results of a subquery, for example, $\pi_{\underline{a}_1}\underline{r}$ would return only attribute $\underline{a}_1$ from $\underline{r}$. A *selection* enables filtering the tuples returned by a subquery based on a given condition $\underline{\theta}$, for example,

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

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

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

$\sigma_{\underline{a}_1 > 3}\underline{r}$ would return all tuples from $\underline{r}$ where the value for $\underline{a}_1$ is greater than 3. A *Cartesian products* simply cross products every tuple from its left subquery with every tuple from its right subquery. The *join* operation joins two subqueries based on a condition and omitting its condition implies it is a natural join (i.e., join on the shared attribute of the two subqueries). For example, $\underline{r}_1 \bowtie_{\underline{a}_1 = \underline{a}_2} \underline{r}_2$ joins tuples from $\underline{r}_1$ and $\underline{r}_2$ where the attribute $\underline{a}_1$ from relation $\underline{r}_1$ is equal to attribute $\underline{a}_2$ from relation $\underline{r}_2$. However, if we have $\underline{r}_1(\underline{a}_1, \underline{a}_3)$ and $\underline{r}_2(\underline{a}_1, \underline{a}_2)$ then $\underline{r}_1 \bowtie \underline{r}_2$ joins tuples from $\underline{r}_1$ and $\underline{r}_2$ where attribute $\underline{a}_1$ has the same value in $\underline{r}_1$ and $\underline{r}_2$. Also, note that join is simply a syntactic sugar for selection of cross product, that is $\underline{q}_1 \bowtie_{\underline{\theta}} \underline{q}_2 = \sigma_{\underline{\theta}}(\underline{q}_1 \times \underline{q}_2)$. The set operations, union and intersection, require two subqueries to have the same set of attributes and simply apply the operation, either union or intersection, to the tuples returned by the subqueries. For example, if we have $\underline{r}_1(\underline{a}_1, \underline{a}_2)$ with tuples $\{(1,2)(3,4)\}$ and $\underline{r}_2(\underline{a}_1, \underline{a}_2)$ with tuples $\{(1,2),(5,6)\}$ then $\underline{r}_1 \cup \underline{r}_2$ returns the tuples $\{(1,2),(3,4),(5,6)\}$.

3.2 Formula Choice Calculus and Annotation

The choice calculus [46, 22] is a metalanguage for describing variation in programs and its elements such as data structures [47, 23]. In the choice calculus, variation is represented in-place as choices between alternative subexpressions. For example, the the variational expression $e = A\langle 1,2\rangle + B\langle 3,4\rangle + A\langle 5,6\rangle$ contains three choices. Each choice has an associated *dimension*, which is used to synchronize the choice with other choices in different parts of the expression. For example, expression e contains two dimensions, A and B, and the two choices in dimension A are synchronized. Therefore, the variational expression e represents four different plain expressions, depending on whether the left or right alternatives are selected from each dimension. Assuming that dimensions evaluate to boolean values, we have: (1) $1+2+5$, A and B evaluate to `true`, (2) $1+4+5$, A evaluates to `true` and B evaluates to `false`, (3) $2+3+6$, A evaluates to `false` and B evaluates to `true`, and (4) $2+4+6$, A and B evaluate to `false`.

The formula choice calculus [28] extends the choice calculus by allowing dimensions to be propositional formulas. For example, the variational expression $e' = A \vee B\langle 1,2\rangle$ represents two plain expressions: (1) 1, $A \vee B$ evaluate to `true` and (2) 1, $A \vee B$ evaluate to `false`.

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

4 Research Goals and Approaches

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

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

Table 2: Objective 1 research questions.

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

For RQ1.1 we explored different application domains where multiple database instances exists simultaneously or over time and they conceptually represent the same or similar data, but, differ slightly in their schema and/or content for business or experimental reasons. We introduced some of these application domains in Section 1.

After investigating the application domains we realized that variation in databases is similar to variation in software systems with regards to the dimensions of variation Software systems can vary in two dimensions: "space" and "time". Variation over space refers to the simultaneous development and maintenance of related systems with different feature sets, which is the focus of work on SPLs. Variation over time refers to the incremental evolution of a system as it is developed and maintained, and is the focus of revision control systems and configuration management (CM) [20]. Although these two aspects of variation have traditionally been studied and addressed separately, recent work has sought to unify the treatment of them to support new kinds of analyses that consider both dimensions of variation at once in order to ease maintenance, support the reuse of techniques developed in their respective communities, and to support new kinds of analyses that consider both dimensions of variation at once [45].

Similarly, variation in databases exists in time and space. Variation in time appears when a database evolves over time while variation in space appears due to different information requirements by software or different sources of information. However, these variation dimensions in databases have not been studied. Instead, as we mentioned in Section 1, instances of them have been studied and addressed separately. For example, schema evolution is an instance of variation in time while variation appearing in database of the software when developing software systems using a SPL is variation in space. However, software evolution is unavoidable, so is its artifacts evolution, including databases [25]. 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 4.1.1 and we use this example throughout this proposal to elaborate more on the introduced concepts.

For RQ1.2 we introduce a *feature space* that captures aspects of variation that may appear in a variational database scenario. A feature is basically some characteristic that is important to the database administrator and developers while interacting with the database. We then introduce *feature expressions* as propositional formulas of features to have an expressive representation of variation. Section 4.1.2 provides the formal definition and examples of features and feature expressions.

For RQ1.3 we provide two case studies by taking two real-world scenarios where variation appears in databases and encode their variation using feature expressions. In our first case study, we focus on variation in "space". It shows the use of feature expression to encode the variation needed for a database-backed SPL. We consider an email SPL which has been used in several previous SPL research projects (e.g. [4, 3]). Since the SPL has a set of features we use the same feature space to account for variation in the database of the software. Our case study is formed by systematically combining two pre-existing works:

1. We use Hall's decomposition of an email system into its component features [24] as high-level specification of a SPL.

2. We use the Enron email dataset[1] as a realistic email database.

In our second case study, we focus on variation that occurs in "time". It demonstrates the use of a feature expressions to encode the variation created by an employee database evolution scenario. We systematically adapt an existing database evolution scenario from Moon et al. [35] into a VDB and populate it by a dataset that is widely used in databases research.[2] For this case, we assign a feature to each version of the schema. For example, for the original schema we have the feature V_1 and as the schema is evolving and each time we have a new schema we assign it a new feature. That is, for the second schema we have the feature V_2 and so on. Objective 3 (Section 4.3) demonstrates how these features are used later on.

Section 4.2 discusses the objective 2 and its research questions.

4.1.1 Motivating Example

In this section, we motivate the interaction of two kinds of variation in databases: database-backed software produced by SPL and schema evolution. Consider a SPL that generates management software for companies. The SPL has an optional feature: *edu*, indicating whether a company provides educational means such as courses for its employees[3]. Software variants that disable *edu* (*edu* = `false`) only provide basic functionalities while ones that enable *edu* provide educational functionalities as well as basic ones. Thus, this SPL yields two types of products: basic and educational.

Software produced by this SPL needs a database to store information about employees, but SPL features impacts the database: while basic variants do not need to store any education-related records educational variants do. In practice, SPL developers use only one database for both variant categories [**?**], which can easily be separated by using the *edu* feature. We visualize this idea in Table 3 with two schema types: basic and educational. Additionally, we introduce *temporal features* to tag schemas when they change. A basic schema is associated with a temporal feature of $V_1 - V_5$ while an educational schema is associated with

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

[2] `https://github.com/datacharmer/test_db`

[3] In practice, such a SPL would have two features: *base* and *edu*, where *base* is an arbitrary feature, i.e., for all variants it must be enabled, and it indicates software variants that provide just basic functionalities. For simplicity and without loss of generality, we drop this feature.

Table 3: Employee schema evolution of a database for a SPL. A feature (a boolean variable) represents inclusion/exclusion of tables/attributes.

Temporal Features	Schemas of Databases for SPL Software Variants		Temporal Features
	basic	educational	
V_1	*engineerpersonnel* (*empno*, *name*, *hiredate*, *title*, *deptname*) *otherpersonnel* (*empno*, *name*, *hiredate*, *title*, *deptname*) *job* (*title*, *salary*)	*course* (*coursename*, *teacherno*) *student* (*studentno*, *coursename*)	T_1
V_2	*empacct* (*empno*, *name*, *hiredate*, *title*, *deptname*) *job* (*title*, *salary*)	*course* (*courseno*, *coursename*, *teacherno*) *student* (*studentno*, *courseno*)	T_2
V_3	*empacct* (*empno*, *name*, *hiredate*, *title*, *deptno*) *job* (*title*, *salary*) *dept* (*deptname*, *deptno*, *managerno*) *empbio* (*empno*, *sex*, *birthdate*)	*course* (*courseno*, *coursename*) *teach* (*teacherno*, *courseno*) *student* (*studentno*, *courseno*, *grade*)	T_3
V_4	*empacct* (*empno*, *hiredate*, *title*, *deptno*, *std*, *instr*) *job* (*title*, *salary*) *dept* (*deptname*, *deptno*, *managerno*) *empbio* (*empno*, *sex*, *birthdate*, *name*)	*ecourse* (*courseno*, *coursename*) *course* (*courseno*, *coursename*, *time*, *class*) *teach* (*teacherno*, *courseno*) *student* (*studentno*, *courseno*, *grade*)	T_4
V_5	*empacct* (*empno*, *hiredate*, *title*, *deptno*, *std*, *instr*, *salary*) *dept* (*deptname*, *deptno*, *managerno*, *stdnum*, *instrnum*) *empbio* (*empno*, *sex*, *birthdate*, *firstname*, *lastname*)	*ecourse* (*courseno*, *coursename*, *deptno*) *course* (*courseno*, *coursename*, *time*, *class*, *deptno*) *teach* (*teacherno*, *courseno*) *take* (*studentno*, *courseno*, *grade*)	T_5

a temporal feature of $T_1 - T_5$. We have two sets of temporal features because when *edu* is enabled any educational and basic schemas can be grouped to form a complete schema. For example, a valid software variant can have the basic schema associated with V_3 and the educational schema associated with T_4.

Now, consider the following scenario: in the initial design of the basic database, SPL database administrators (DBAs) settle on three tables *engineerpersonnel*, *otherpersonnel*, and *job*; shown in Table 3 and associated with temporal feature V_1. After some time, they decide to refactor the schema to remove redundant tables, thus, they combine the two relations *engineerpersonnel* and *otherpersonnel* into one, *empacct*; associated with temporal feature V_2. Since some of clients' software relies on a previous design the two schemas have to coexist in parallel. Therefore, the existence (presence) of *engineerpersonnel* and *otherpersonnel* relations is *variational*, i.e., they only exist in the basic schema when V_1 = `true`. This scenario describes *component evolution*: database evolution in SPL resulted from developers update, refactor, improve, and components [25].

Now, consider the case where a client that previously requested a basic variant of the management software has recently added courses to educate its employees in subjects they need. Hence, the SPL needs to enable the *edu* feature for this client, forcing the adjustment of the schema to educational. This case describes *product evolution*: database evolution in SPL resulted from clients adding/removing features/-components [25].

The two basic and educational schemas are not independent of each other: consider the basic schema variant for temporal feature V_4. Attributes *std* and *instr* only exists in the *empacct* relation when *edu* = `true`, represented by dash-underlining them, otherwise the *empacct* relation has only four attributes: *empno*, *hiredate*, *title*, and *deptno*. Hence, the presence of attributes *std* and *instr* in *empacct* relation is *variational*, i.e., they only exist in *empacct* relation when *edu* = `true`.

Our motivating example demonstrates how variation in time and space arises and how they interact,

an unavoidable consequence of modern software development. To be concrete throughout the proposal we itemize the needs of users working with a database with variation through the needs of SPL developers and DBAs:

(N0) SPL developers need to access all database variants while writing code to be able to extract information for all software variants they are developing. Hence, *users need to have access to all database variants at a given time*.

(N1) Depending on what component they are working on, they need to be able to query all or some of the variants. Hence, *users need to query multiple database variants simultaneously and selectively*.

(N2) Given that users have access to all database variants and that they can query all variants simultaneously, for test purposes, they desire to know which variant a tuple belongs to. Hence, *the framework needs to keep track of which variants a piece of data belongs to and ensuring that it is maintained throughout a query*.

(N3) SPL developers need to deploy the database and its queries to generate a specific software product for a client based on their requested features. Hence, *users need to deploy one variant of the database and its associated queries*.

Current solutions to database variation not only cannot adapt to a new kind of variation but also cannot satisfy all these needs. Current SPLs generate and use messy databases by employing a universal schema. However, this burdens DBAs heavily because they need to write specific queries for each variant in order to avoid getting messy data, thus, current SPLs cannot satisfy **N1** and **N2**. As stated in Section 1, schema evolution systems only consider evolution in time and do not provide any means for **N1**-**N3**. Also, the current solution to the problem of schema evolution within a SPL is addressed by designing a new domain-specific language so that SPL developers can write scripts of the schema changes [25], which still requires a great effort by DBAs and SPL developers and it does not address **N0**-**N2** sufficiently.

4.1.2 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 **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 3); for SPLs, the features can be adopted from the SPL feature set (e.g. the *edu* feature in our motivating example, Table 3); 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 **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 **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 3, 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 $\{V_2, T_3, edu\}$.

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:

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

Figure 2: Feature expression syntax and relations.

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

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

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

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

4.2 Design and implement a database framework that accommodate identified variations

Having an encoding that represent variation we need to incorporate it within the database and the query language to allow explicit storing and manipulation of variation in a database. Objective 2 aims to design and implement a database framework that considers variation as a first-class citizen. Table 4 presents individual research questions we need to answer for this objective.

Table 4: Objective 2 research questions.

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

For RQ2.1 we annotate elements of a database with feature expression. We use annotated elements both in the schema and content. Within a schema we allow attributes and relations to exist conditionally based on the feature expression assigned to them, called *variational schema*. Section 4.2.2 provides the formal definition of variational schema and examples of it. At the content level, we annotate each tuple with a feature expression, indicating when the tuple is present, called *variational table*. Section 4.2.3 provides the formal definition and examples of variational table which uses *variational sets*, introduced in Section **??**, for the formalizations. Together variational schema and a set of variational tables associated with the variational schema create a *variational database (VDB)*, defined in Section 4.2.3. Conceptually, a *single* variational database represents *many* different plain relational databases, each one corresponding to a different variant of a software system, *at the same time*.

For RQ2.2 we need to design a query language that reflects the variational information need while working with a variational database. To express and represent variation in queries, we incorporate formula choice calculus [28] into a structured query language. We formally define *variational relational algebra (VRA)* in Section 4.2.4 as our algebraic query language. A query written in VRA is called a *variational query (v-query)*; when it is clear from context we use query and v-query interchangeably. A v-query typically conveys the same intent over several relational database variants, however, a single v-query is also capable of capturing different intents over different database variants.

For RQ2.3 and to interact with VDBs using v-queries, we are implementing and optimizing *Variational Database Management System (VDBMS)*. VDBMS is implemented in Haskell. VDBMS sit on top of any DBMS that the user desires and used to store their data in form of variational tables. To support running VDBMS with multiple different plain relational DBMS backends, we provide a shared interface for connecting to and inquiring information from a DBMS and instantiate it for different database engines such as PostgreSQL and MySQL. An expert can extend VDBMS to another database engine by writing methods for connecting to and querying from the database. Section 4.2.5 provides details of our implementation and the architectur of VDBMS.

Section 4.3 discusses the objective 3 and its research questions.

4.2.1 Variational Set

A *variational set (v-set)* $X = \{x_1{}^{e_1}, \ldots, x_n{}^{e_n}\}$ is a set of annotated elements [23, 47, 7]. 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* $\{2^{f_1}, 3^{f_1}, 4\} \not\prec \{2, 3^{f_1 \wedge f_2}\}$.

4.2.2 Variational Schema

A variational schema captures variation in the structure of a database by indicating which attributes and relations are included or excluded in which variants. To achieve this we annotate attributes, relations, and

the schema itself with feature expressions, which describe the condition under which they are present. A *variational relation schema (v-relation schema)*, s, is a relation name accompanied with an annotated variational set of attributes: $s \in \mathbf{R} ::= r(A)^e$. The presence condition of the v-relation schema, e, determines the set of all possible relation schema variants for relation r. A *variational schema (v-schema)* is an annotated set of v-relation schemas: $S \in \mathbf{S} ::= \{s_1, \ldots, s_n\}^m$. The presence condition of the v-schema, m, determines all configurations for non-empty schema variants. We call such configurations *valid* configurations. The v-schema's presence condition is the VDB feature model since it captures the relationship between features of the underlying application and their constraints, Hence, the v-schema defines all valid schema variants of a VDB.

Example 6. *S_1 is the v-schema of a VDB including only relations empacct and ecourse in the last two rows of Table 3, where only features are V_4, V_5, edu, T_4, T_5. Note that attributes that exist conditionally are annotated with a feature expression to account for such a condition, e.g., the salary attribute only exists when V_5 = `true`.*

$$\begin{aligned} S_1 = \{&empacct(empno, hiredate, title, deptno, salary^{V_5}, std^{edu}, instr^{edu})^{V_4 \vee V_5} \\ &, ecourse(courseno, coursename, deptno^{T_5})^{T_4 \vee T_5}\}^{m_1} \\ m_1 = &(\neg edu \wedge (V_4 \oplus V_5)) \vee (edu \wedge (V_4 \oplus V_5) \wedge (T_4 \oplus T_5)) \end{aligned}$$

where m_1 allows only one temporal feature for each schema column be enabled at a given time.

The presence of an attribute naturally depends on the presence of its parent v-relation, which in turn depends on the feature model. Thus, the presence condition of an attribute is the conjunction of its presence condition with its v-relation's presence condition and the feature model. Similarly, the presence condition of a v-relation is the conjunction of its presence condition and the feature model. In other words, the annotated attribute a^e of v-relation r with $e_r = pc(r)$ defined in the v-schema S with feature model m is valid if: $sat(e \wedge e_r \wedge m)$. For example, the *std* attribute described in Example 6 is only valid if its presence condition is satisfiable, i.e., $sat(edu \wedge (V_4 \vee V_5) \wedge m)$.

In essence, a v-schema is a systematic compact representation of all schema variants of the underlying application of interest. A specific pure relational schema for a database variant can be obtained by *configuring* the v-schema with that variant's configuration. We define the configuration function for v-schemas and its elements in Figure 3. For example, consider the v-schema in Example 6. Configuring the variational attribute set of the *empacct* v-relation for the variant $\{V_5\}$, i.e., $\mathbb{A}[\![empno, hiredate, title, deptno]\!]_{\{V_5\}}$, yields the attribute set of $\{\underline{empno}, \underline{hiredate}, \underline{title}, \underline{deptno}\}$.

4.2.3 Variational Table

Variation also exists in database content. To account for content variability, we tag tuples with presence conditions, e.g., the tuple $(1,2)^{f_1}$ only exists when f_1 is enabled. Thus, a *variational tuple (v-tuple)* is an annotated tuple where there is a mapping between the v-relation attribute list and tuple's values: $u \in U ::= (v_1, \ldots, v_l)^{e_u}$ where the a relation schema is $r(a_1, \ldots, a_l)^{e_r}$. The content of a v-relation is a set of v-tuples $U \in \mathbf{T} ::= \{u_1, \ldots, u_k\}$ and a *variational table (v-table)* is the pair of its relation schema and content: $t = (s, U)$. A *variational database instance*, $\mathscr{I}_S$, of VDB D with v-schema S, is a set of v-tables: $\mathscr{I} \in \mathbf{I} ::= \{t_1, \ldots, t_n\}$.

Note that the value v_i is present iff $sat(e_u \wedge e_r \wedge e_a \wedge m)$, where, $e_a = pc(att(v)_i)$ and for simplicity, we only annotate tuples and not cells. This design decision causes some redundancy. For example, the two tuples: $(1,2)^{f_1}$ and $(1,3)^{\neg f_1}$ cannot be represented as a single tuple $(1, f_1\langle 2,3\rangle)$ that has variation at its cell-level.

This encoding of variational databases satisfies all of the needs for a variational database described in Section 4.1.1. Similar to v-schema, a user can configure a v-table or a VDB for a specific variant, formally defined in Figure 3. This allows users to deploy a VDB for a specific configuration (variant), satisfying database part of **N3** need. Additionally, our VDB framework puts all variants of a database into one VDB (satisfying **N0**) and it keep tracks of which variant a tuple belongs to by annotating them with presence conditions. For example, consider tuples $(38, PL, 678)^{T_5}$ and $(23, DB, \texttt{NULL})^{T_4}$ that belong to the *ecourse* table. The presence conditions T_5 and T_4 state that tuples belong to variants four and five of this VDB, respectively. Hence, this framework tracks which variants a tuple belongs to (first part of **N2**). Note that the VDB framework encodes both schematic and content-level variation. A simpler framework could be used to encode only content-level variation (where tables consist of v-tuples but they have plain relational schema), similar to frameworks used for database versioning and experimental databases [27]. However, schematic variation cannot be encoded without accounting for content-level variation in a framework where variants coexist in parallel and they are all put into one database, e.g., while $ecourse\left(courseno, coursename, deptno^{T_5}\right)^{edu \wedge (T_4 \vee T_5)}$ encodes variation at the schema-level for relation *ecourse*, dropping presence conditions of given tuples (i.e., resulting in tuples $(38, PL, 678)$ and $(23, DB, \texttt{NULL})$) where it is unclear which variant each tuple belongs to and there is no way to recover such information.

4.2.4 Variational Relational Algebra

To account for variation, VRA combines relational algebra (RA) with *choices* [46, 22]. A choice $e\langle x, y\rangle$ consists of a feature expression e and two alternatives x and y. For a given configuration c, the choice $e\langle x, y\rangle$ can be replaced by x if e evaluates to `true` under configuration c, (i.e., $\mathbb{E}[\![e]\!]_c$), or y otherwise. In essence, choices allow a v-queries to encode variation in a structured and systematic manner.

Variational conditions:

$$\theta \in \Theta \quad ::= \quad b \;\mid\; \underline{a} \bullet k \;\mid\; \underline{a} \bullet \underline{a} \;\mid\; \neg\theta \;\mid\; \theta \vee \theta \;\mid\; \theta \wedge \theta \;\mid\; e\langle \theta, \theta \rangle$$

Variational relational algebra syntax:

$$\begin{array}{rcll} q \in \mathbf{Q} & ::= & r & \textit{Variational relation} \\ & \mid & \sigma_\theta q & \textit{Variational selection} \\ & \mid & \pi_A q & \textit{Variational projection} \\ & \mid & e\langle q, q \rangle & \textit{Choice of queries} \\ & \mid & q \times q & \textit{Variational cartesian product} \\ & \mid & q \circ q & \textit{Variational set operation} \\ & \mid & \varepsilon & \textit{Empty relation} \end{array}$$

Figure 4: Variational relational algebra definitions. $\bullet$ and $\circ$ denote comparison $(<, \leq, =, \neq, >, \geq)$ and v-set operations $(\cap, \cup)$, respectively. We consider *variational join* $q_1 \bowtie_\theta q_2$ as syntactic sugar for $\sigma_\theta(q_1 \times q_2)$.

The syntax of VRA is given in Figure 4. In VRA, the selection operation is similar to standard RA selection except that the condition parameter is *variational* meaning that it may contain choices. For example, the query $\sigma_{e\langle a_1 = a_2, a_1 = a_3 \rangle} r$ selects a v-tuple u if it satisfies the condition $a_1 = a_2$ and $sat(e \wedge pc(u))$ or if $a_1 = a_3$ and $sat(\neg e \wedge pc(u))$. The projection operation is parameterized by a v-set of attributes. For example, the

Variational Set of Attributes Configuration:

$$\mathbb{A}[\![.]\!] : \mathbf{A} \rightarrow \mathbf{C} \rightarrow \underline{\mathbf{A}}$$

$$\mathbb{A}[\![\{a^e\} \cup A]\!]_c = \begin{cases} \{\underline{a}\} \cup \mathbb{A}[\![A]\!]_c, & \text{if } \mathbb{E}[\![e \wedge pc(rel(a)) \wedge m]\!]_c \\ \mathbb{A}[\![A]\!]_c, & \text{otherwise} \end{cases}$$

$$\mathbb{A}[\![\{\}]\!]_c = \{\}$$

Variational Relation Schema Configuration:

$$\mathbb{R}[\![.]\!] : \mathbf{R} \rightarrow \mathbf{C} \rightarrow \underline{\mathbf{R}}$$

$$\mathbb{R}[\![r(A)^e]\!]_c = \begin{cases} r(\mathbb{A}[\![A]\!]_c, & \text{if } \mathbb{E}[\![e \wedge m]\!]_c) \\ \varepsilon, & \text{otherwise} \end{cases}$$

Variational Schema Configuration:

$$\mathbb{S}[\![.]\!] : \mathbf{S} \rightarrow \mathbf{C} \rightarrow \underline{\mathbf{S}}$$

$$\mathbb{S}[\![\{r_1(A_1)^{e_1}, \ldots, r_n(A_n)^{e_n}\}^m]\!]_c = \begin{cases} \{\mathbb{R}[\![r_1(A_1)^{e_1 \wedge m}]\!]_c, \ldots, \mathbb{R}[\![r_n(A_n)^{e_n \wedge m}]\!]_c\}, & \text{if } \mathbb{E}[\![m]\!]_c \\ \{\}, & \text{otherwise} \end{cases}$$

Variational Tuple Configuration:

$$\mathbb{U}[\![.]\!] : U \rightarrow \mathbf{C} \rightarrow \underline{U}$$

$$\mathbb{U}[\![(v_1, \ldots, v_l)^{e_u}]\!]_c = (\mathbb{V}[\![v_1]\!]_c, \ldots, \mathbb{V}[\![v_l]\!]_c) \quad \textit{where } \forall 1 \leq i \leq l : \mathbb{V}[\![v_i]\!]_c = \begin{cases} v_i, & \text{if } \mathbb{E}[\![m \wedge pc(rel(att(v_i))) \wedge pc(att(v_i)) \wedge e_u]\!]_c \\ \varepsilon, & \text{otherwise} \end{cases}$$

V-Relation Content Configuration:

$$\mathbb{T}[\![.]\!] : \mathbf{T} \rightarrow \mathbf{C} \rightarrow \underline{\mathbf{T}}$$
$$\mathbb{T}[\![\{u_1, \ldots, u_k\}]\!]_c = \{\mathbb{U}[\![u_1]\!]_c, \ldots, \mathbb{U}[\![u_k]\!]_c\}$$

VDB Instance Configuration:

$$\mathbb{I}[\![.]\!] : \mathbf{I} \rightarrow \mathbf{C} \rightarrow \underline{\mathbf{I}}$$
$$\mathbb{I}[\![\{t_1, \ldots, t_n\}]\!]_c = \{(\mathbb{R}[\![s_1]\!]_c, \mathbb{T}[\![U_1]\!]_c), \ldots, (\mathbb{R}[\![s_n]\!]_c, \mathbb{T}[\![U_n]\!]_c)\}$$

Figure 3: V-cond and VDB instance configurations. ε denotes a non-existent relation and value.

query $\pi_{a_1,a_2{}^e} r$ projects a_1 from relation r unconditionally, and a_2 when $sat(e)$. The choice operation enables combining two v-queries to be used in different variants based on a given feature expression. In practice, it is often useful to return information in some variants and nothing at all in others. We introduce an explicit *empty* query ε to facilitate this. The empty query is used, for example, in q_2 in Example 7. The rest of VRA's operations are similar to RA, where all set operations (union, intersection, and cross product) are changed to the corresponding variational set operations defined in Section 4.2.1. In examples, we also use a join operation with a variational condition, $q_1 \bowtie_\theta q_2$, which is syntactic sugar for $\sigma_\theta(q_1 \times q_2)$.

Our implementation of VRA also provides mechanisms for renaming queries and qualifying attributes with relation/subquery names. These features are needed to support self joins and projecting attributes with the same name in different relations. However, for simplicity, we omit these features from the formal definition in this proposal.

The result of a v-query is a v-table with the relation name *result*. For example, assume that v-tuples $(1,2)^{f_1}$ and $(3,4)^{\neg f_3}$ belong to a v-relation $r(a_1,a_2)$, which is the only relation in a VDB with the trivial feature model `true`. The query $f_3\langle\pi_{a_1{}^{f_2}} r, \varepsilon\rangle$ returns a v-table with relation schema $result(a_1{}^{f_2})^{f_3}$, which indicates that the result is only non-empty when f_3 is true and that the result includes attribute a_1 when f_2 is true. Section 4.4.2 defines a type system that yields the relation schema for any well-formed query. The content of the result relation is a single v-tuple $(1)^{f_1}$. The tuple $(3)^{\neg f_3}$ is not included since the projection occurs in the context of the choice in f_3, which is incompatible with the presence condition of the tuple (i.e., $unsat(f_3 \wedge \neg f_3)$). This illustrates how choices can effectively filter the tuples in a VDB based on their presence conditions.

The following example illustrates, in the context of our running example, how a v-query can be used to express variational information needs.

Example 7. *Assume a VDB with features V_3, V_4, and V_5, and the corresponding empbio schema variants in Table 3. The v-schema for this VDB is:*

$$S_2 = \{empbio(empno, sex, birthdate, name^{V_4}, firstname^{V_5}, lastname^{V_5})\}^{m_2}$$
$$where\ m_2 = V_3 \oplus V_4 \oplus V_5.$$

The user wants the employee ID numbers (empno) and names for variants V_4 and V_5. The user needs to project the name attribute for variant V_4, the firstname and lastname attributes for variant V_5, and empno attribute for both variants. This can be expressed with the following v-query.

$$q_1 = \pi_{empno^{V_4 \vee V_5}, name, firstname, lastname} empbio$$

Note that the user does not need to repeat the variability information encoded in the v-schema in their query, that is, they do not need to annotate *name*, *firstname*, and *lastname* with V_4, V_5, and V_5, respectively. We discuss this in more detail in Section 4.4.2 and Section 4.4.3. q_1 queries all three variants simultaneously although the returned results are only associated with variants V_4 and V_5 due to the annotation of the attribute *empno* in the query and the presence conditions of the rest of the projected attributes in the schema. Yet, selecting only two out of the three variants can be written more explicitly in a query by using a choice: $q_2 = V_4 \vee V_5\langle\pi_{empno,name,firstname,lastname} empbio, \varepsilon\rangle$. Note that queries q_1 and q_2 return the same set of v-tuples since neither returns tuples associated with variant V_3, but their returned v-tables have different presence conditions, as will be discussed later in Example 11.

VRA has semantic-preserving equivalence rules that allow users to incorporate their taste and preference of where and how they want to encode variation in their queries. These rules factor out the commonality of subqueries and generate queries with less variation. Section 4.4.5 expands on these rules.

The next example illustrates how a v-query can be used to express the same intent over several database variants using choices and conditions. Expressing the same intent over several instances by a single query relieves the DBA from maintaining separate queries for different versions or configurations of the schema.

Example 8. *Assume a VDB with features V_1–V_5 and the corresponding* basic *schema variants in Table 3. The user wants to get all employee names across all variants, which they can express by the following v-query.*

$$q_3 = V_1\langle \pi_{name} engineerpersonnel \cup \pi_{name} otherpersonnel, \\ (V_2 \vee V_3)\langle \pi_{name} empacct, (V_4 \vee V_5)\langle \pi_{name,firstname,lastname} empbio, \varepsilon\rangle\rangle\rangle$$

Since the v-schema enforces that exactly one of V_1–V_5 be enabled, we can simplify the query by omitting the final choice.

$$q_4 = V_1\langle \pi_{name} engineerpersonnel \cup \pi_{name} otherpersonnel, (V_2 \vee V_3)\langle \pi_{name} empacct, \pi_{name,firstname,lastname} empbio\rangle$$

In principle, v-queries can also express arbitrarily different intents over different database variants. However, we expect that v-queries are best used to capture single (or at least related) intents that vary in their realization since this is easier to understand and increases the potential for sharing in both the representation and execution of a v-query.

The semantics of VRA can be understood as a combination of the *configuration semantics* of VRA, defined in Figure 5, the configuration semantics of VDBs, defined in Figure 3, and the semantics of plain RA. Thus, the v-query semantics is the set of semantics of its configured relational queries over their corresponding configured relational database variant for every valid configuration of the feature model of the VDB. The configuration function maps a v-query under a given configuration to a pure relational query, defined in Figure 5. Users can deploy queries for a specific variant by configuring them, satisfying query part of **N3** need stated in Section 4.1.1.

Example 9. *Assume the underlying VDB has the v-schema $S_3 = \{r\left(a_1{}^{f_1}, a_2, a_3\right)^{f_1 \vee f_2}\}$ and only two features f_1 and f_2. The v-query $q_5 = \pi_{a_1, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2}} r$ is configured to the following relational queries: $\mathbb{Q}[\![q_5]\!]_{\{f_1\}} = \mathbb{Q}[\![q_5]\!]_{\{\,\}} = \pi_{\underline{a}_1} \underline{r}$, $\mathbb{Q}[\![q_5]\!]_{\{f_2\}} = \pi_{\underline{a}_1, \underline{a}_3} \underline{r}$, $\mathbb{Q}[\![q_5]\!]_{\{f_1, f_2\}} = \pi_{\underline{a}_1, \underline{a}_2, \underline{a}_3} \underline{r}$.*

VRA enables querying multiple database variants encoded as a singled VDB simultaneously and selectively, satisfying the query need state in Section 4.1.1 (**N1**). More precisely, VRA is *maximally expressive* in the sense that it can express any set of plain RA queries over any subset of relational database variants encoded as a VDB. This claim is captured by the following theorem.

Theorem 10. *Given a set of plain RA queries $\underline{q}_1, \ldots, \underline{q}_n$ where each query $\underline{q}_i$ is to be executed over a disjoint subset $\mathscr{I}_i$ of variants of the VDB instance $\mathscr{I}$, there exists a v-query q such that $\forall c.\, \mathbb{I}[\![\mathscr{I}]\!]_c = \mathscr{I}_i \implies \mathbb{Q}[\![q]\!]_c = \underline{q}_i$.*

Proof. By construction. Let f_i be the feature expression that uniquely characterizes the variants in each $\mathscr{I}_i$. Then $q =$
$(f_1 \wedge \neg f_2 \wedge \ldots \wedge \neg f_n)\langle \underline{q}_1, (f_2 \wedge \ldots \wedge \neg f_n)\langle \underline{q}_2, \ldots f_n\langle \underline{q}_n, \varepsilon\rangle \ldots\rangle\rangle$.

□

The above construction relies on the fact that every RA query is a valid VRA (sub)query in which every presence condition is true. Of course, in most realistic scenarios, we expect that v-queries can be encoded more efficiently by sharing commonalities and embedding relevant choices and presence conditions within the v-query.

Variational Condition Configuration:

$$
\begin{aligned}
\mathbb{C}[\![.]\!] &: \Theta \to \mathbf{C} \to \underline{\Theta} \\
\mathbb{C}[\![b]\!]_c &= b \\
\mathbb{C}[\![\underline{a} \bullet k]\!]_c &= \underline{a} \bullet k \\
\mathbb{C}[\![\underline{a}_1 \bullet \underline{a}_2]\!]_c &= \underline{a}_1 \bullet \underline{a}_2 \\
\mathbb{C}[\![\neg\theta]\!]_c &= \neg\mathbb{C}[\![\theta]\!]_c \\
\mathbb{C}[\![\theta_1 \vee \theta_2]\!]_c &= \mathbb{C}[\![\theta_1]\!]_c \vee \mathbb{C}[\![\theta_2]\!]_c \\
\mathbb{C}[\![\theta_1 \wedge \theta_2]\!]_c &= \mathbb{C}[\![\theta_1]\!]_c \wedge \mathbb{C}[\![\theta_2]\!]_c \\
\mathbb{C}[\![e\langle\theta_1,\theta_2\rangle]\!]_c &= \begin{cases} \mathbb{C}[\![\theta_1]\!]_c, & \text{if } \mathbb{E}[\![e]\!]_c \\ \mathbb{C}[\![\theta_2]\!]_c, & \text{otherwise} \end{cases}
\end{aligned}
$$

Variational Query Configuration:

$$
\begin{aligned}
\mathbb{Q}[\![.]\!] &: \mathbf{Q} \to \mathbf{C} \to \underline{\mathbf{Q}} \\
\mathbb{Q}[\![r]\!]_c &= \mathbb{R}[\![r]\!]_c = \underline{r} \\
\mathbb{Q}[\![\sigma_\theta q]\!]_c &= \sigma_{\mathbb{C}[\![\theta]\!]_c}\mathbb{Q}[\![q]\!]_c \\
\mathbb{Q}[\![\pi_A q]\!]_c &= \pi_{\mathbb{A}[\![A]\!]_c}\mathbb{Q}[\![q]\!]_c \\
\mathbb{Q}[\![q_1 \times q_2]\!]_c &= \mathbb{Q}[\![q_1]\!]_c \times \mathbb{Q}[\![q_2]\!]_c \\
\mathbb{Q}[\![e\langle q_1,q_2\rangle]\!]_c &= \begin{cases} \mathbb{Q}[\![q_1]\!]_c, \text{ if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \mathbb{Q}[\![q_2]\!]_c, \text{ otherwise} \end{cases} \\
\mathbb{Q}[\![q_1 \circ q_2]\!]_c &= \mathbb{Q}[\![q_1]\!]_c \circ \mathbb{Q}[\![q_2]\!]_c \\
\mathbb{Q}[\![\varepsilon]\!]_c &= \underline{\varepsilon}
\end{aligned}
$$

Figure 5: Configuration of VRA which assumes that the given v-query is well-typed. Note that we have extended RA with an empty relation $\underline{\varepsilon}$.

4.2.5 VDBMS Architecture

Figure 6 shows the architecture of VDBMS and its modules. The implantation of VDB directly follows its formalization given in Section 4.2.3. The only difference is that the presence conditions of a tuple is encoded as an extra attribute in the underlying relational database. For now, we assume a VDB and its v-schema are generated by an expert and are stored in a DBMS, we provide guidelines and case studies of systematically generating VDBs in Section A. A VDB can be *configured* to its pure relational database variants, if desired by a user, by providing the configuration of the desired variant, Figure 3. For example, a SPL developer configures a VDB to produce software and its database for a client.

Given a VDB and its v-schema, a user inputs a v-query q to VDBMS. First, q is checked by the *type system* to determine if it is invalid, explained in Section 4.4.2. If so, the user gets errors explaining what part of the query violated the v-schema. Otherwise, q is explicitly annotated with the schema, defined in Section 4.4.3, to ensure variation-preserving property w.r.t. v-schema throughout the execution flow of v-query in the system and then it is passed to the *variation minimization* module, introduced in Section 4.4.5, to minimize the variation of q and apply relational algebra optimization rules. The optimized query is then sent to the *generator* module where SQL queries are generated from v-queries using one of the following approaches:

1. *Naive Brute Force (NBF):* Configures a v-query q for all valid configurations, i.e., $\forall c \in \mathbf{C}$, translates them to RA queries, and finally generates SQL queries with renaming all subqueries. The SQL queries are sent to underlying DBMS and the results are gathered and cleaned up in v-table builder module. Here is the flow of how results are generated by this approach:

$$q \overset{\mathbb{Q}[\![q]\!]_c}{\rightarrow} [(c, \underline{q})] \underset{\text{to SQL}}{\overset{\text{translate}}{\rightarrow}} [(c, \underline{sql})] \underset{\text{queries}}{\overset{\text{run}}{\rightarrow}} \textit{v-tables} \underset{\text{builder}}{\overset{\text{v-table}}{\rightarrow}} \textit{v-table}$$

2. *Unique Brute Force (UBF):* This approach is just like NBF except that we only generate SQL queries for unique RA queries which are the result of configuring the v-query. The trick is to attach the condition under which a RA query is valid. This condition is a feature expression generated from the set of configurations that configured the v-query into the same RA query, i.e., $\mathsf{Q}(q) = \{\underline{q}^e \mid \forall c \in \mathbf{C} : \mathbb{E}[\![e]\!]_c = \texttt{true}, \mathbb{Q}[\![q]\!]_c = \underline{q}\}$. The implementation of this function is more efficient than its definition. That is:

$$q \overset{\mathsf{Q(q)}}{\rightarrow} [\underline{q}^e] \underset{\text{to SQL}}{\overset{\text{translate}}{\rightarrow}} [\underline{sql}^e] \underset{\text{queries}}{\overset{\text{run}}{\rightarrow}} \textit{v-tables} \underset{\text{builder}}{\overset{\text{v-table}}{\rightarrow}} \textit{v-table}$$

3. *Union-All-Variants (UAV):* This approach takes the SQL queries generated by UBF and unions them to just run one SQL query. In order to do so it forces all the SQL queries to return the same set of attributes. Additionally, it applies the presence condition of a SQL query to its tuples by concating it with the presence condition attribute in the projected attribute set. The query is sent to the underlying DBMS and result is cleaned up and returned to the user as a v-table. Note that cleaning up the result is part of v-table builder tasks. That is:

$$q \overset{\mathsf{Q(q)}}{\rightarrow} [\underline{q}^e] \underset{\text{SQL}}{\overset{\text{generate}}{\rightarrow}} [\underline{sql}^e] \overset{\text{union}}{\rightarrow} \underline{sql}' \underset{\text{query}}{\overset{\text{run}}{\rightarrow}} \textit{v-table} \overset{\text{clean up}}{\rightarrow} \textit{v-table}$$

We use these SQL generators to simulate current approaches used to manage variation currently. In particular, NBF simulates how a naive DBA would conduct such a task. UBF simulates how an expert DBA would conduct such a task. The generated SQL queries need to be independent from the underlying DBMS

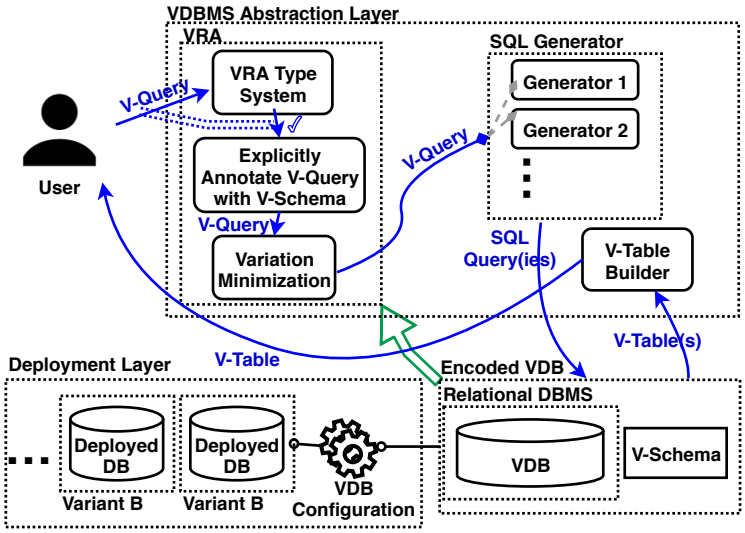


Figure 6: VDBMS architecture and execution flow of a v-query. The dotted double-line from v-query to pushing v-schema module indicates the dependency of passing the v-query to this module only if it is valid. The dashed gray arrows with diamond heads demonstrate an option for the flow of input. The blue filled arrows track the data flow, the green hollow arrows indicate an input to a module.

that stores the VDB. Hence, the SQL generator module has a submodule that prints generated SQL queries for each DBMS engine.

Having generated SQL queries, they are now run over the underlying VDB (stored in a DBMS desired by the user). The result could be either a v-table or a list of v-tables, depending on the approach chosen in the translator to RA and SQL generator modules. The v-table(s) is passed to the *v-table builder* to create one v-table that filters out duplicate and invalid tuples, shrinks presence conditions, and eventually, returns the final v-table to the user.

4.3 Demonstrate how the proposed system manages variation in databases in different application domains and how effective and efficient it is

Having a variational database framework, we need to examine how effectively it represents instances of variation in databases in different application domains. Objective 3 focuses on this goal and Table 5 represents individual research question we need to answer for this objective.

Table 5: Objective 3 research questions.

Objective 3: Demonstrate how the proposed system can be used to manage variation in databases in different application domains
RQ3.1: Can real-world instances of variation in databases be encoded as a VDB and can their variational information need be queried by VRA? What are the steps to generate a VDB from a scenario of variation in a database? (VaMoS'21)
RQ3.2: What are the benefits and drawbacks of representing variation generically in databases as opposed to having scenario-tailored approaches? What are possible improvements in the future?(VaMoS'21)
RQ3.3: How effective and efficient is VDBMS? (VLDB'21)

For RQ3.1 we develop the two case studies discussed in Section 4.1 as variational databases with a set of variational queries that capture a variety of their variational information needs. Section A provides the systematic approach we took to generate these databases and Section B provides their set of queries.

Section A and Section B illustrate steps of explicitly encoding variation in the database and queries. The first step is recognizing features involved in the variational scenario. A feature is a trait that is essential for convey some information need. Thus, for each feature, one should (1) enumerate the operations that must be supported both to implement the feature itself and to resolve undesirable feature conflicts, (2) identified the information needs to implement these operations, and (3) map features to database elements accordingly. This process outlines generating the v-schema for a variational scenario as show cased for our case studies in Section A.2.2 and Section A.1.2. Note that we generated our VDBs from scratch, that is, we did not have any database variants and their corresponding configuration. It is easy to generate a VDB from a set of database variants with their corresponding configuration.

Generating the case studies provides us with an insight for RQ3.2. We expected our language to be more useable, readable, and easier to understand. However, we realize that there is a buy-in cost for programmers to pay when they first learn the language. Furthermore, the variation in queries may get complicated, yet the programmer can still express their information need the way they are most comfortable since variability flows in different parts of a query as shown in Section B.2, that is, the overall query can be wrapped in a choice, attributes can be projected variationally and conditions can be encoded variationally. Still, we believe that the expressiveness of our language allows for type checking queries (with variation encoded in them) even though it adds more complexity to the language.

The case studies provide an accessible example of how one can encode different kinds of variability in a database using a VDB. The capability to encode variation directly in databases allows one to have a starting point for attacking a new kind of variability appearing in databases, for example, one could have used the employee case study to attack database evolution without developing an entirely new system. We suspect that VDBs can be used to also model database versioning [11, 27] and data provenance [14]. Such applications can be pursued in future. Furthermore, our encoding can be used to emulate the encoding of variation within other sorts of data such as spreadsheets.

Encoding variation explicitly in both the database and the query provides developers with tracing variation and connecting the variation in data to variation in software development. For example, the ability to check properties over data as mentioned in Section 4.4.1 is really useful for debugging the human mistakes. Similarly, the type system ensures that queries do not contain undesired behavior. Thus, we believe that explicitly encoding variability in the data and the queries gives developers the power of testing properties that otherwise would not be possible. This is arguably the strongest advantage of VDBs. Recently lots of

attempts have been done on statically analysis, theorem proving, model checking, and program verification of variable programs and variable structure [12, 13, 32, 44], especially for safety-critical systems, and since databases are part of software system the ability to analyze, check, and verify properties over variational databases provides programmers with a seamless framework where they can verify software and its artifacts.

For RQ3.3 we are currently evaluating VDBMS with our case studies and we plan to compare our different implementation approaches in terms of performance.

Section 4.4 discusses the objective 4 and its research questions.

4.4 Mechanize proofs of properties of the language and the system

Having established that our framework effectively and explicitly encode variation within the database and its query language, we need to ensure that it satisfies the properties we desire. Objective 4 aims to define such properties and prove that they hold for our framework. Table 6 presents individual research questions we need to answer for this objective.

Table 6: Objective 4 research questions.

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

For RQ4.1 we identified properties that must hold for a variational database to ensure the variation has encoded correctly in both the schema and content. Section 4.4.1 provides these properties and discusses how a database administrator can write their own properties over a variational database and ensure that it holds.

For RQ4.2 we pay attention to the variational nature of VRA. Consequently, the expressiveness of v-queries may cause them to be more complicated than relational queries. Hence, we introduce a *type system* for VRA that statically checks if a v-query conforms to the underlying v-schema and encoded variability within the VDB. VRA's type system is formally defined in Section 4.4.2.

Additionally, similar to other applications of variational research [17, 16], the type system must preserve the variation encoded in a v-query. We call this property *variation preserving* with respect to the variational schema and formally define it in Section 4.4.4. In order for the variation-preserving property to hold we need to define a function that decouples a query from the underlying variational schema. We achieve this by *explicitly annotating* a query with the variational schema. This function is formally defined in Section 4.4.3. We are currently proving this property mechanically using the Coq proof assistant.

Moreover, since variation points in a v-query can move throughout the query it is important to provide a set of rules that minimize the variation in a v-query. We provide some of these rules in Section 4.4.5. We plan to mechanically prove that these equivalence rules are correct. We also plan to define the formal semantics of VRA to prove that the variation-preserving property also holds at the semantic level to ensure that queries behave as expected.

For RQ4.3 we plan to use the the formal semantics of VRA to prove that the implementation of VRA in VDBMS aligns with the semantics of it.

Section 4.5 discusses a stretch goal that we like to pursue if time permits.

4.4.1 Property Checking over a VDB

Since a *single* database can supply data for *many* different database variants *at the same time* encoding variability explicitly in a database allows the developers to check for different properties over all database variants For example, to ensure that variability associated with each variant is valid we provide a set of validity checks. These checks ensure that the presence conditions both at the schema level and data level are consistent and satisfiable, that is, they are present in at least one database variant. In the following, the function $sat(e)$ denotes a satisfiability check that returns `true` if the feature expression e is satisfiable and `false` otherwise.

At the schema level we check the following properties:

1. That there is at least one valid configuration of the feature model m: $sat(m)$
2. That every relation r is present in at least one configuration of the variational schema: $\forall r \in S, sat(m \wedge pc(r))$
3. That 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. That 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 we check the following properties:

1. That 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. That 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 case studies described in Section A 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., the old data is also included in a more recent variant in addition to the newly added data, it is desirable to ensure that older variants are subsets of newer variants. Assume that configurations $c_1, c_2, \cdots$ represents time-orderly configurations, we formulated and checked this for the employee use case: $\forall c_i, c_j \in \mathbf{C}, i \leq j, [\![D]\!]_{c_i} \subseteq [\![D]\!]_{c_j}$, where $[\![D]\!]_c$ denotes configuring a variational database instance D for configuration c.

4.4.2 Well-Typed (Valid) Query

To prevent running v-queries that have errors we implement a *static type system* for VRA. The type system ensures queries are *well-typed*, i.e., they comply with the underlying v-schema, both w.r.t. the traditional structure of the database and the variability encoded in the database. Assume we have the VDB given in Example 9 with the only relation $r\left(a_1{}^{f_1}, a_2, a_3\right)^{f_1 \vee f_2}$. Attribute a_4 cannot be projected from r because it is not present in r, thus, the query $\pi_{a_4} r$ is invalid. Similarly, the query $\pi_{a_1{}^{\neg f_1}} r$ has an error because a_1 is not present in r for $\forall c \in \mathbf{C}. \mathbb{E}[\![\neg f_1]\!]_c = \texttt{true}$, but these are the only configurations where the query desires to project attribute a_1 from r.

Figure 7 defines VRA's *typing relation* as a set of inference rules assigning *types* to queries. The type

V-queries typing rules:

$$\textsc{EmptyRelation-E}\quad e,S \vdash \varepsilon : \{\,\}^{\texttt{false}} \qquad \textsc{Relation-E}\quad \frac{r(A)^{e'} \in S \qquad sat(e \wedge e')}{e,S \vdash r : A^{e \wedge e'}}$$

$$\textsc{Project-E}\quad \frac{e,S \vdash q : A'^{e'} \qquad |\downarrow(A^e)| = |A| \qquad A \prec \downarrow\left(A'^{e'}\right)}{e,S \vdash \pi_A q : \left(A \cap A'\right)^{e'}} \qquad \textsc{Select-E}\quad \frac{e,S \vdash q : A^{e'} \qquad e, \downarrow\left(A^{e'}\right) \vdash \theta}{e,S \vdash \sigma_\theta q : A^{e'}}$$

$$\textsc{Choice-E}\quad \frac{e \wedge e', S \vdash q_1 : A_1^{e_1} \qquad e \wedge \neg e', S \vdash q_2 : A_2^{e_2}}{e,S \vdash e'\langle q_1, q_2\rangle : \left(\downarrow\left(A_1^{e_1}\right) \cup \downarrow\left(A_2^{e_2}\right)\right)^{(e_1 \wedge e') \vee (e_2 \wedge \neg e')}} \qquad \textsc{Product-E}\quad \frac{e,S \vdash q_1 : A_1^{e_1} \qquad e,S \vdash q_2 : A_2^{e_2} \qquad \downarrow\left(A_1^{e_1}\right) \cap \downarrow\left(A_2^{e_2}\right) = \{\}}{e,S \vdash q_1 \times q_2 : \left(\downarrow\left(A_1^{e_1}\right) \cup \downarrow\left(A_2^{e_2}\right)\right)^{e_1 \wedge e_2}}$$

$$\textsc{SetOp-E}\quad \frac{e,S \vdash q_1 : A_1^{e_1} \qquad e,S \vdash q_2 : A_2^{e_2} \qquad \downarrow\left(A_1^{e_1}\right) \equiv \downarrow\left(A_2^{e_2}\right)}{e,S \vdash q_1 \circ q_2 : A_1^{e_1}}$$

V-condition typing rules (b: boolean tag, $\underline{a}$: plain attribute, k: constant value):

$$\textsc{Conjunction-C}\quad \frac{e,A \vdash \theta_1 \qquad e,A \vdash \theta_2}{e,A \vdash \theta_1 \wedge \theta_2} \qquad \textsc{Disjunction-C}\quad \frac{e,A \vdash \theta_1 \qquad e,A \vdash \theta_2}{e,A \vdash \theta_1 \vee \theta_2} \qquad \textsc{Choice-C}\quad \frac{e \wedge e', A \vdash \theta_1 \qquad e \wedge \neg e', A \vdash \theta_2}{e,A \vdash e'\langle \theta_1, \theta_2\rangle}$$

$$\textsc{Neg-C}\quad \frac{e,A \vdash \theta}{e,A \vdash \neg\theta} \qquad \textsc{AttOptVal-C}\quad \frac{a^{e'} \in A \qquad sat\left(e' \wedge e\right) \qquad k \in dom_{\mathscr{I}}(a)}{e,A \vdash \underline{a} \bullet k} \qquad \textsc{Boolean-C}\quad e,A \vdash b$$

$$\textsc{AttOptAtt-C}\quad \frac{{a_1}^{e_1} \in A \qquad {a_2}^{e_2} \in A \qquad sat(e_1 \wedge e_2 \wedge e) \qquad type(a_1) = type(a_2)}{e,A \vdash \underline{a}_1 \bullet \underline{a}_2}$$

Figure 7: VRA and v-condition typing relation. The typing rule of a join query is the combination of rules SELECT-E and PRODUCT-E.

of a query is a v-relation schema $result(A)^e$, however, for brevity and since the relation name is the same for all queries we consider the type of a query an annotated v-set of attributes where attributes are projected by the query from the VDB and their presence conditions determine their valid variants. The presence condition of attributes in the type of a query may differ from their presence conditions in v-schema due to variation constraints imposed by the query. For example, continuing with relation $r\left(a_1{}^{f_1}, a_2, a_3\right)^{f_1 \vee f_2}$, the query $\pi_{a_2{}^{f_1}} r$ has the type $\{a_2{}^{f_1}\}^{f_1 \vee f_2}$ while according to r's schema $pc(a_2) = f_1 \vee f_2$, i.e., the presence condition of attribute a_2 changes through the query. The presence condition of the entire set determines the condition under which the entire table (i.e., attributes and tuples) are valid. Note that it is essential to consider the type of a query an *annotated* v-set to account for the presence condition of the entire table.

VRA's typing relation, as defined in Figure 7, has the judgement form $e, S \vdash q : A^{e'}$. This states that in *variation context* e within v-schema S, v-query q has type $A^{e'}$. If a query does not have a type, it is *ill-typed*. *Variation context* is a feature expression that the type system keeps and refines to keep track of variation encoded by a query. The variation context is initiated by the feature model. For brevity, we use the judgment form $S \vdash q : A^{e'}$ for $pc(S), S \vdash q : A^{e'}$ i.e., the variation context is initialized. Note that attributes with an unsatisfiable presence condition are not present in any database variant, i.e., they are not present for any configuration. Thus, the existence of such attribute in a type does not change the type semantically, based on the defined equivalence rule for v-sets, given in Definition 4. Hence, we do not filter out such attributes explicitly in Figure 7, however, for simplicity, the implemented type system drops the attributes with an unsatisfiable presence condition.

The rule RELATION-E states that, in variation context e with underlying variational schema S, assuming that 1) S contains the relation r with presence condition e' and v-set of attributes A and 2) there exists a valid variant in the intersection of variation context e and r's presence condition e', i.e., $sat(e \wedge e')$, then query r has type A annotated with $e \wedge e'$.

The rule PROJECT-E states that, in variation context e within v-schema S, assuming that the subquery q has type $A'^{e'}$, v-query $\pi_A q$ has type $(A \cap A')^{e'}$, if all attributes in A are present in e and $\downarrow\left(A'^{e'}\right)$ subsumes A. The subsumption, defined in Definition 5, ensures that the subquery q does not have an empty type and it includes all attributes in the projected attribute set and attributes' presence conditions do not contradict each other. Returning the intersection of types, defined in Definition 2, filters both attributes and their presence conditions. Example 11 illustrates generating the type of a query step by step.

Example 11. *We illustrate how a query enforces variation encoded within it to the result. We do this by illustrating how the type system generates two different types for queries q_1 and q_2 given in Example 7. For brevity, we simplify feature expressions when possible. For q_1, it applies the* PROJECT-E *rule under the variation context initiated to $m_2 = V_3 \oplus V_4 \oplus V_5$ and schema S_2. It now has to apply the* RELATION-E *rule to the subquery empbio under the same variation context and schema, resulting in the type $A_{empbio} = \{empno, sex, birthdate, name^{V_4}, firstname^{V_5}, lastname^{V_5}\}^{m_2}$. Now that it has the type of the subquery empbio it verifies that the projected attribute v-set $A_{prj} = \{empno^{V_4 \vee V_5}, name, firstname, lastname\}^{m_2}$, is subsumed by A_{empbio}. Thus, it generates the type of query q_1 by intersecting A_{prj} and A_{empbio} annotated with A_{empbio}'s presence condition resulting in the type $A_{q_1} = \{empno^{V_4 \vee V_5}, name^{V_4}, firstname^{V_5}, lastname^{V_5}\}^{m_2}$. This type demonstrates the structure of the result of query q_1. As for q_2, the type system applies the* CHOICE-E *rule under the variation context initiated to m_2 and schema S_2. It then applies the* PROJECT-E *and* EMPTYRELATION-E *rules to the left and right alternatives of the choice, respectively, which generates the types $A_{left} = (empno, name^{V_4}, firstname^{V_5}, lastname^{V_5})^{m_2 \wedge (V_4 \vee V_5)}$ and $A_{right} = \{\}^{\texttt{false}}$, respectively. Finally, it generates the type of q_2 by annotating the union of A_{left} and A_{right} with $m_2 \wedge (V_4 \vee V_5)$, resulting in the final type of*

$A_{q_2} = (empno, name^{V_4}, firstname^{V_5}, lastname^{V_5})^{m_2 \wedge (V_4 \vee V_5)}$. *Note that* A_{q_2}*'s presence condition explicitly accounts for only two variants while* A_1 *does not do so even though* q_1 *does not return any tuple that belong to variant* $\{V_3\}$ *because of its attributes presence condition.*

The rule SELECT-E states that, in variation context e within v-schema S, assuming that the subquery q has type $A^{e'}$, the v-query $\sigma_\theta q$ has type $A^{e'}$, if the variational condition θ is well-formed w.r.t. variation context e and type $A^{e'}$, denoted by v-condition's typing relation $e, \downarrow\left(A^{e'}\right) \vdash \theta$. Note that in variational condition typing rules, the presence condition of the query type is pushed in. The rules state that attributes used in a variational condition must be valid in A and attribute's presence condition e' in type A must exists within variation context e, denoted by $sat(e' \wedge e)$. They also check the constraints of traditional relational databases, such as the type of two compared attributes must be the same.

The rule CHOICE-E states that, in variation context e within v-schema S, the type of a choice of two subqueries is the *union of types*, defined in Definition 1, of its subqueries annotated with the disjunction of their presence conditions conjuncted with the corresponding condition of the choice's dimension. A choice query is well-typed iff both of its subqueries q_1 and q_2 are well-typed. Note that CHOICE-E is the only rule that refines the variation context.

The rule PRODUCT-E states that the type of a product query in variation context e is the union of the type of its subqueries annotated with the conjunction of their presence conditions, assuming that they are disjoint.

The rule SETOP-E denotes the typing rule for set operation queries such as union and difference. It states that, if the subqueries q_1 and q_2 have types $A_1^{e_1}$ and $A_2^{e_2}$, respectively, in variation context e, then the v-query of their set operation has type $A_1^{e_1}$, iff $\downarrow(A_1^{e_1})$ and $\downarrow(A_2^{e_2})$ are *equivalent*. The *type equivalence* is v-set equivalence, defined in Definition 4, for v-sets of attributes.

4.4.3 Explicitly Annotating Queries

V-queries do not need to repeat information that can be inferred from the v-schema or the type of a query. For example, the query q_1 shown in Example 7 does not contradict the schema and thus is type correct. However, it does not include the presence conditions of attributes and the relation encoded in the schema while q_5 repeats this information:

$$q_5' = \pi_{empno^{V_4 \vee V_5}, name^{V_4}, firstname^{V_5}, lastname^{V_5}}(m_2\langle \pi_{empno, sex, birthdate, name^{V_4}, firstname^{V_5}, lastname^{V_5}} empbio, \varepsilon \rangle)$$

Similarly, the outer projection in the query $q_6 = \pi_{name, firstname} V_4 \langle \pi_{name} q_5, \pi_{firstname} q_5 \rangle$ written over S_2 does not repeat the presence conditions of attributes from its subquery's type. The query

$$q_7 = \pi_{a^{V_4} name, firstname^{\neg V_4}} V_4 \langle \pi_{name} q_5, \pi_{firstname} q_5 \rangle$$

makes the annotations of projected attributes *explicit* w.r.t. both the v-schema S_2 and its subquery's type. Although relieving the user from explicitly repeating variation makes VRA easier to use, queries still have to state variation explicitly to avoid losing such information when decoupled from the schema. We do this by defining a function, $\lfloor q \rfloor_S$, with type $\mathbf{Q} \to \mathbf{S} \to \mathbf{Q}$, that *explicitly annotates a query q given the underlying schema S*. Note that $\lfloor q \rfloor_S$ needs to take the underlying schema as an input since it is using the type system (which relies on the schema) as a helper function. The explicitly annotating query function, formally defined in Figure 8, conjuncts attributes and relations presence conditions with the corresponding annotations in the query and wraps subqueries in a choice when needed. Queries q_7 and q_5 are examples of applying the explicitly annotation function to queries q_6 and q_1, respectively, after simplifying them.

$$
\begin{aligned}
\lfloor . \rfloor_S &: \mathbf{Q} \to \mathbf{S} \to \mathbf{Q} \\
\lfloor r \rfloor_S &= pc(r)\langle \pi_A r, \varepsilon \rangle \quad \textit{where} \quad S \vdash r : A \\
\lfloor \sigma_\theta q \rfloor_S &= \sigma_\theta \lfloor q \rfloor_S \\
\lfloor \pi_A q \rfloor_S &= \pi_{A \cap A'} \lfloor q \rfloor_S \quad \textit{where} \quad S \vdash \pi_A \lfloor q \rfloor_S : A' \\
\lfloor q_1 \times q_2 \rfloor_S &= \lfloor q_1 \rfloor_S \times \lfloor q_2 \rfloor_S \\
\lfloor e\langle q_1, q_2 \rangle \rfloor_S &= e\langle \lfloor q_1 \rfloor_{\downarrow(S^e)}, \lfloor q_2 \rfloor_{\downarrow(S^{\neg e})} \rangle \\
\lfloor q_1 \circ q_2 \rfloor_S &= \lfloor q_1 \rfloor_S \circ \lfloor q_2 \rfloor_S \\
\lfloor \varepsilon \rfloor_S &= \varepsilon
\end{aligned}
$$

Figure 8: Explicitly annotating queries with the underlying v-schema. Queries passed to this function are well-typed.

Theorem 12. *If the query q has the type A then its explicitly annotated counterpart has the same type A, i.e.:*

$$S \vdash q : A \Rightarrow S \vdash \lfloor q \rfloor_S : A' \textit{ and } A \equiv A'$$

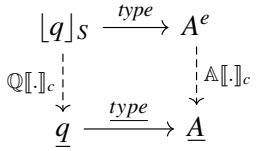
This shows that the type system applies the schema to the type of a query although it does not apply it to the query[4].

We illustrate the application of Theorem 12 for queries q_1 and q_5. Example 11 explained how q_1 is generated step-by-step. The variation context and underlying schema are the same and the subquery *empbio* has the same type. The projected attribute set annotated with the variation context is: $A_2 = \{empno^{V_4 \vee V_5}, name^{V_4}, firstname^{V_5}, lastname^{V_5}\}^{m_2}$, which is clearly subsumed by A_{empbio}, thus, its intersection with A_{empbio} annotated with the presence condition of A_{empbio} is itself, which makes it obvious that $A_{q_1} \equiv A_{q_5}$.

4.4.4 Variation-Preserving Property with respect to. Schema

Similar to other applications of variational research [17, 16], the type system must preserve the variation encoded in a v-query. We define the *variation-preserving property with respect to v-schema*: if a query q has type A then configuring the type of a valid explicitly annotated query is the same as the type of its configured corresponding query. Theorem 13 defines this property formally.

In the diagram, the vertical arrows indicate corresponding configure functions, *type* indicates VRA's type system, i.e., $type(q) = A^e$ is $S \vdash q : A^{e'}$, and $\underline{type}$ indicates RA's type system, i.e., $\underline{S} \vdash \underline{q} : \underline{A}$. Note that for simplicity, we assume that corresponding v-schema and schema is passed to type systems. Simply put, the relational type of the configured v-query q with configuration c, i.e., $\mathbb{A}[\![type(q)]\!]_c$, must be the same as the configured variational type of the v-query q with configuration c, i.e., $\underline{type}(\mathbb{Q}[\![q]\!]_c)$. *Clearly the diagram commutes*: taking either path of 1) configuring q first and then getting the relational type of it or 2) getting the variational type of q first and then configuring it results in the same set of attributes. The variation-preserving property enforces the maintenance of variants that a tuple belongs to through running a query, satisfying second part of **N2**. Variation-preserving property of VRA's type system



[4]We plan to prove this theorem in the Coq proof assistant.

and RA's type safety [36] implies that VRA's type system is also type safe. Example 14 illustrates why the query must be explicitly annotated with the v-schema in the variation-preserving diagram.

Theorem 13. *For all configurations c, if a query q has type A then its configured query* $\mathbb{Q}[\![\lfloor q \rfloor_S]\!]_c$ *has type* $\mathbb{A}[\![A]\!]_c$*, i.e.,*

$$\forall c \in \mathbf{C}.S \vdash q : A \Rightarrow \mathbb{S}[\![S]\!]_c \vdash \mathbb{Q}[\![\lfloor q \rfloor_S]\!]_c : \mathbb{A}[\![A]\!]_c \quad .$$

Example 14. *Consider the v-query* $q_5 = \pi_{a_1{}^{f_1}, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2} r}$ *given in Example 9. It is well-typed and it has the type* $A = \{a_1{}^{f_1}, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2}\}$*. Configuring A for the variant that both* f_1 *and* f_2 *are disabled results in an empty attribute set. However, the type of its configured query for this variant, i.e.,* $\mathbb{Q}[\![q_5]\!]_{\{\,\}} = \pi_{\underline{a}_1 \underline{r}}$*, is the attribute set* $\{\underline{a}_1\}$*. This violates the variation-preserving property. A similar problem happens for the variant of* $\{f_2\}$*, i.e.,* $\underline{type}\left(\mathbb{Q}[\![q_5]\!]_{\{f_2\}}\right) = \underline{type}\left(\pi_{\underline{a}_1, \underline{a}_3 \underline{r}}\right) = \{\underline{a}_1, \underline{a}_3\} \neq \{\underline{a}_3\} = \mathbb{A}[\![A]\!]_{\{f_2\}} = \mathbb{A}[\![type\,(q_5)]\!]_{\{f_2\}}$*. However, the variation-preserving property holds for the explicitly annotated query with the v-schema, i.e.,* $\lfloor q_5 \rfloor_{S_3} = \pi_{a_1{}^{f_1}, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2} r}$*.*

4.4.5 Variation Minimization

VRA is flexible since an information need can be represented via multiple v-queries as demonstrated in Example 7 and Example 8. It allows users to incorporate their personal taste and task requirements into v-queries they write by having different levels of variation. For example, consider the explicitly annotated query q_5 in Section 4.4.3:
$q_5 = \pi_{empno^{V_4 \vee V_5}, name^{V_4}, firstname^{V_5}, lastname^{V_5}} m_2\langle empbio, \varepsilon \rangle$ To be explicit about the exact query that will be run for each variant the user can *lift up* the variation and rewrite the query as
$q'_5 = V_4\langle \pi_{empno,name} empbio, V_5\langle \pi_{empno,firstname,lastname} empbio, \varepsilon \rangle \rangle$. While q_5 contains less redundancy q'_5 is more comprehensible. Thus, *supporting multiple levels of variation creates a tension between reducing redundancy and maintaining comprehensibility.*

We define *variation minimization* rules, Figure 9. Pushing in variation into a query, i.e., applying rules left-to-right, reduces redundancy while lifting them up, i.e., applying rules right-to-left, makes a query more understandable. When applied left-to-right, the rules are terminating since the scope of variation monotonically decreases in size.

Choice Distributive Rules:

$$
\begin{aligned}
e\langle \pi_{A_1} q_1, \pi_{A_2} q_2 \rangle &\equiv \pi_{A_1^e, A_2^{\neg e}} e\langle q_1, q_2 \rangle \\
e\langle \sigma_{\theta_1} q_1, \sigma_{\theta_2} q_2 \rangle &\equiv \sigma_{e\langle \theta_1, \theta_2 \rangle} e\langle q_1, q_2 \rangle \\
e\langle q_1 \times q_2, q_3 \times q_4 \rangle &\equiv e\langle q_1, q_3 \rangle \times e\langle q_2, q_4 \rangle \\
e\langle q_1 \bowtie_{\theta_1} q_2, q_3 \bowtie_{\theta_2} q_4 \rangle &\equiv e\langle q_1, q_3 \rangle \bowtie_{e\langle \theta_1, \theta_2 \rangle} e\langle q_2, q_4 \rangle \\
e\langle q_1 \circ q_2, q_3 \circ q_4 \rangle &\equiv e\langle q_1, q_3 \rangle \circ e\langle q_2, q_4 \rangle
\end{aligned}
$$

CC and RA Optimization Rules Combined:

$$
\begin{aligned}
e\langle \sigma_{\theta_1 \wedge \theta_2} q_1, \sigma_{\theta_1 \wedge \theta_3} q_2 \rangle &\equiv \sigma_{\theta_1 \wedge e\langle \theta_2, \theta_3 \rangle} e\langle q_1, q_2 \rangle \\
\sigma_{\theta_1} e\langle \sigma_{\theta_2} q_1, \sigma_{\theta_3} q_2 \rangle &\equiv \sigma_{\theta_1 \wedge e\langle \theta_2, \theta_3 \rangle} e\langle q_1, q_2 \rangle \\
e\langle q_1 \bowtie_{\theta_1 \wedge \theta_2} q_2, q_3 \bowtie_{\theta_1 \wedge \theta_3} q_4 \rangle &\equiv \sigma_{e\langle \theta_2, \theta_3 \rangle} (e\langle q_1, q_3 \rangle \bowtie_{\theta_1} e\langle q_2, q_4 \rangle)
\end{aligned}
$$

Figure 9: Some of variation minimization rules.

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

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

4.6 Summary

Figure 10 summarizes the connections between the research questions and activities. Table 4.6 provides the timeline for this proposal.

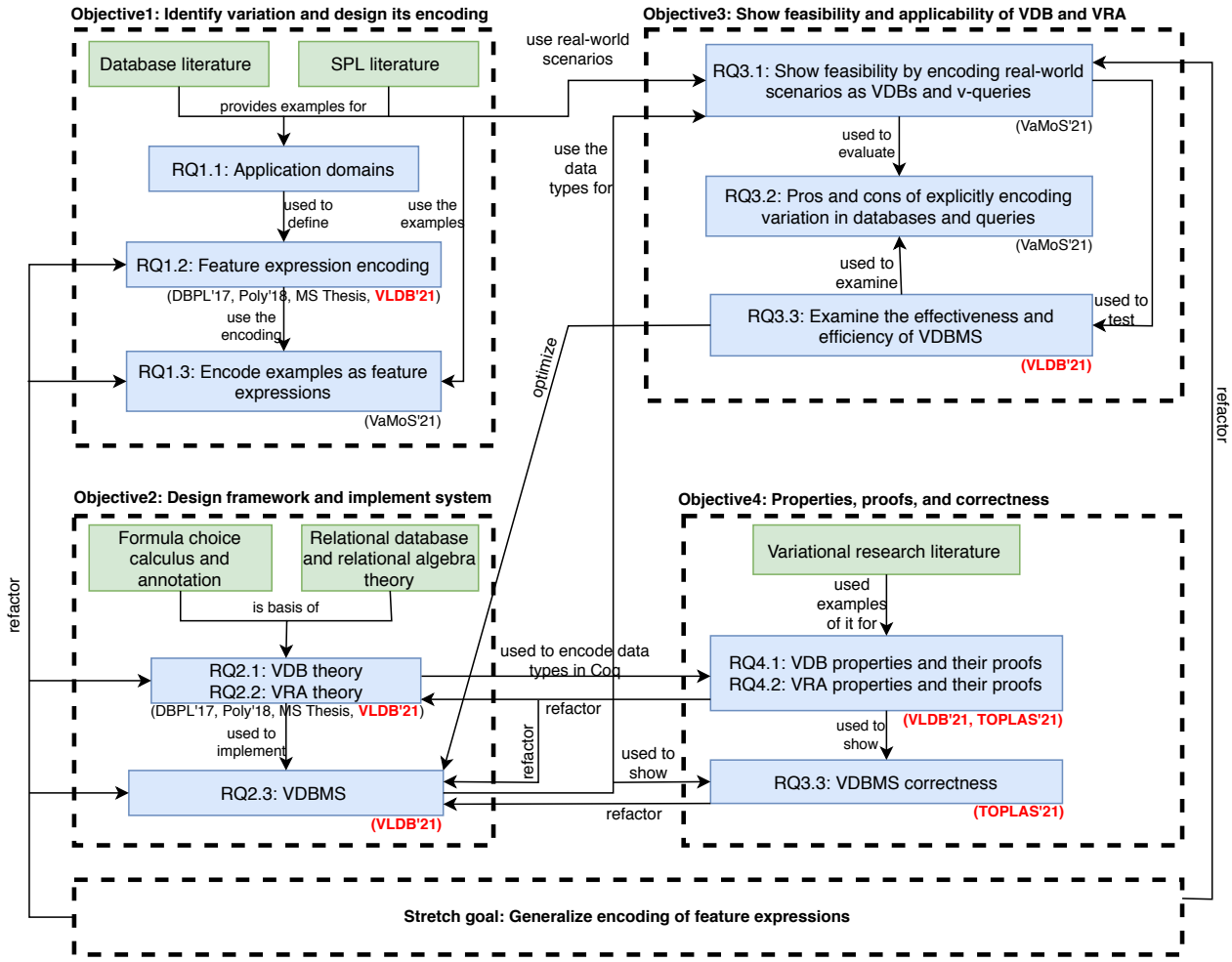


Figure 10: The connection between objectives and research questions. The bolden publications in red are the ones that haven't been submitted yet.

Table 7: Summary of projected and completed dates for each of the proposed research questions.

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

5 Related Work

Software product line: The SPL community has a tradition of developing and distributing case studies to support research on software variation. For example, SPL2go [43] 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 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. 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 [31] address data model variability in the context of delta-oriented programming. They define delta modules that can incrementally generate a relational database schema, and so can be used to generate different database schemas for each variant of a SPL. Humblet et al. [29] 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. [39] generates variable database schema from a given global schema and software variants configurations by mapping schema elements to features. Siegmund et al. [41] emphasizes the need for a variable database schema in SPL and proposes two decomposition approaches: (1) physical where database sub-schemas associated with a feature are stored in physical files and (2) virtual where a global Entity-Relation model of a schema is annotated with features. All of these approach address the issue of *structural* database variation in SPLs and provide a technique to derive a database schema per variant, which is also achievable by configuring a VDB. The work of Humblet et al. [29] is most similar to our notion of a variational schema since it is an annotative approach [30] that enables directly associating schema elements with features. Abo Zaid and De Troyer [2] is also annotative, but operates at the higher level of a data model that may only later be realized as a relational database. Khedri and Khosravi [31] is a compositional approach [30] to generating database schemas. None of these approaches consider *content-level* variation, which is captured by VDBs and observable in our case studies, nor do they consider how to express queries over databases with structural variation, which is addressed by our *variational queries*.

While the previous approaches all address data variation in space, Herrmann et al. [25] 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.

Variational databases, originally designed to represent the intrinsic variability of databases used in SPL [8], explicitly account for variation in databases by considering a set of features for a database that encodes its variability. Unlike other approaches taken in SPL introduced at the beginning of this section, it collapses all the database variants into one variational database while tracing the variation by annotating elements with feature expressions [7]. The variational database management system allows users to query all database variants, i.e., the overall variational database, simultaneously and selectively [9]. It also allows users to deploy the variational database (similar to approaches described briefly above) and variational queries for a specific variant (while mentioned approaches are unable to achieve this).

Instances of variation in database literature: 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* [19, 35, 26, 37], 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.

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 [35] or as versions attached to databases [33, 15, 6, 42], 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 [26], 2) they require the DBA to specify the version for their query and then migrating all database variants to the queried version [33, 15, 6, 42], or 3) they require the user to specify the timestamps for their query and then reformulate the query for other database variants [35].

These approaches usually do not grant users access to old variants of data even if they desire so and it is messy to keep both different copies of a variant, one with the old schema and one with the unified schema, since every data addition/update now requires to be applied to all copies of the database variant. A better solution is maintaining a history of the changes applied to the database and the unified schema as an XML document and providing a language that allows users/developers to choose the variant they desire [35]. Unfortunately, this is achieved by limiting the schema evolution to temporal changes, offering a beautifully tailored approach for temporal changes, however, resulting in a non-extensible approach for non-temporal changes.

Temporal evolution is tracked by requiring the database to always have a time-related attribute in tables. Thus, queries have to specify the time frame for which they are inquiring information [35]. Now the user can choose a wide enough time frame in their queries to access to their desired variant(s). Aside from the detailed mapping of time frames and variants, this approach requires a query to have one and only one information need, no matter how many variants it is aiming. That is, if a time frame includes assumingely two variants a user cannot write a query that extract two separate information needs for each of them accumulatively in one query. Even worse, if this query does not conform to one of the variant's schema but it conforms to the other one, the query still fails since there is no systematic way to identify that the query is ill (does not conform to the schema) for one of the variants. These limitations and constraint are the result of ignoring that temporal changes to a database is a form of variability.

Work on *database versioning* [11, 27] 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. Work on *data integration* and *database versioning* can be viewed as managing database variation in space [21]. In data integration, the goal is typically to combine data variation from disparate sources and provide a unified interface for querying that data. This is different from VDBs, which make differences between variants explicit, which is needed to manage data variation in SPLs.

The definition of variation is very limited in these problems. Such limitation allows for an efficient intelligent solutions, however, it tailors their solutions to a specific context and prevents one from using the same solution/system in a similar context when variation in time or space appears in a database [38]. For example, one cannot use a data integration system to manage variation in a database used in software produced by a SPL.

Variational research: The representation of v-schemas and variational tables is based on previous work on variational sets [23], which is part of a larger effort toward developing safe and efficient variational data structures [47, 34]. 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 [47]. Although we have focused on variational databases to support SPL development, the broader motivation of *effectively computing with variability* is at the heart of our work. This is why VDBs support not only structural variation but also content-level variation. Also, while variational queries can be statically configured in the same way that SPLs typically are, our prototype VDBMS implementation also supports directly executing variational queries on variational databases to yield variational results.

6 Conclusion

Informed by different instances of variation appearing in databases, we hypothesize that considering variation as an orthogonal concern to database provides benefits researchers, database administrators, and developers. To investigate this hypothesis, we plan the following:

- Activity 1: We have studied various instances of variation in databases and based on them we have provided a framework that considers variation as a first-class citizen in databases [7, 8].
- Activity 2: We have used the framework to represent real-world instances of variation in database, showing the applicability of our frames [10].
- Activity 3: We are implementing and refactoring VDBMS, a database management system for our framework.
- Activity 4: We will mechanically prove the properties of our framework.

The result of these activities will provide the following contributions:

- The first work to establish theoretical framework for variational databases where the database and the query language both allow explicit encoding of variation. (Activity 1 and 4)
- The first database management system that allows developers and database administrators to interact with variational databases. (Activity 1, 2, and 3)

- Real-world case studies of variation in databases that can be used in future research on variational data. (Activity 2)

The database community has researched instances of variation in databases extensively without acknowledging that they are instances of the same problem, variation in a database. On the other hand, the SPL community has realized the need for encoding variation at the data model, however, they do not go beyond the data model. Our work on RQ1.1 and RQ1.3 [10] has shown that different instances of variation in databases could interact with each other, thus, having a generic encoding of variation in databases is beneficial.

References

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

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

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

[4] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *IEEE Int. Conf. on Software Engineering*, pages 482–491, 2013.

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

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

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

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

[9] Parisa Ataei, Qiaoran Li, Eric Walkingshaw, and Arash Termehchy. Managing variability in relational databases by vdbms, 2019.

[10] Parisa Ataei, Qiaoran Li, and Eric Walkingshaw. Should variation be encoded explicitly in databases? In *Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems (submitted)*, VAMOS '21. Association for Computing Machinery, 2021.

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

[12] Tabea Bordis, Tobias Runge, Alexander Knüppel, Thomas Thüm, and Ina Schaefer. Variational correctness-by-construction. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*, VAMOS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375016. doi: 10.1145/3377024.3377038. URL `https://doi.org/10.1145/3377024.3377038`.

[13] Daniel Bruns, Vladimir Klebanov, and Ina Schaefer. Verification of software product lines with delta-oriented slicing. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software*, pages 61–75, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-18070-5.

[14] Peter Buneman and Wang-Chiew Tan. Provenance in databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 1171?1173, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936868. doi: 10.1145/1247480.1247646. URL https://doi.org/10.1145/1247480.1247646.

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

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

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

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

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

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

[21] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 0124160441, 9780124160446.

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

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

[24] Robert J. Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering*, 12 (1):41–79, 2005.

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

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

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

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

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

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

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

[32] Jing Liu, Josh Dehlinger, and Robyn Lutz. Safety analysis of software product lines using state-based modeling. *Journal of Systems and Software*, 80:1879–1892, 11 2007. doi: 10.1016/j.jss.2007.01.047.

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

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

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

[36] Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, page 174–183, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 089791273X. doi: 10.1145/62678.62700. URL https://doi.org/10.1145/62678.62700.

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

[38] John F Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383 – 393, 1995. ISSN 0950-5849. doi: https://doi.org/10.1016/0950-5849(95)91494-K. URL http://www.sciencedirect.com/science/article/pii/095058499591494K.

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

[40] Jitesh Shetty and Jafar Adibi. The Enron Email Dataset: Database Schema and Brief Statistical Report. Technical report, Information Sciences Institute, University of Southern California, 2004.

[41] Norbert Siegmund, Christian Kästner, Marko Rosenmüller, Florian Heidenreich, Sven Apel, and Gunter Saake. Bridging the gap between variability in client application and database schema. In Johann-Christoph Freytag, Thomas Ruf, Wolfgang Lehner, and Gottfried Vossen, editors, *Datenbanksysteme in Business, Technologie und Web (BTW) - 13. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS)*, pages 297–306, Bonn, 2009. Gesellschaft f́ur Informatik e.V.

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

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

[44] Thomas Thüm, Jens Meinicke, Fabian Benduhn, Martin Hentschel, Alexander von Rhein, and Gunter Saake. Potential synergies of theorem proving and model checking for software product lines. volume 1, 09 2014. doi: 10.1145/2648511.2648530.

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

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

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

A Variational Case Studies

A.1 Variation in Space: Email SPL Case Study

In our first case study, we focus 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 which has been used in several previous SPL research projects (e.g. [4, 3]). Our case study is formed by systematically combining two pre-existing works:

1. (1) We use Hall's decomposition of an email system into its component features [24] as high-level specification of a SPL.

Table 8: 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*)

2. (2) We use the Enron email dataset[5] 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.

A.1.1 Variation Scenario: An Email SPL

The email SPL consists of the following features from Hall [24]:

- *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.
- *mailhost*: A list of known users is maintained and known users may retrieve messages on demand. Messages sent to unknown users are rejected.

Note that 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 and they are so closely conceptually related, 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 $m_{en} = \texttt{true}$.

Table 9: V-schema of the email VDB with feature model m_{en}. Presence conditions are colored blue for clarity.

$employeelist(eid, firstname, lastname, email\_id, folder, status, verification\_key^{signature}, public\_key^{encryption})$
$messages(mid, sender, date, message\_id, subject, body, folder, is\_system\_notification, is\_encrypted^{encryption}$
$, is\_autoresponse^{autoresponder}, is\_signed^{signature}, is\_forward\_msg^{forwardmessages})$
$recipientinfo(rid, mid, rtype, rvalue)$
$forward\_msg(eid, forwardaddr)^{forwardmessages}$
$mailhost(eid, username, mailhost)^{mailhost}$
$filter\_msg(eid, suffix)^{filtermessages}$
$remail\_msg(eid, pseudonym)^{remailmessage}$
$auto\_msg(eid, subject, body)^{autoresponder}$
$alias(eid, email, nickname)^{addressbook}$

A.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 8, then systematically adjust its schema to align with the information needs of the email SPL described by Hall [24]. The *employeelist* table contains information about the employees of the company including the employee identification number (*eid*), their first name and last name (*firstname* and *lastname*), their primary email address (*email_id*), alternative email addresses (e.g. *email2*), a path to the folder that contains their data (*folder*), and their last status in the company (*status*). The *messages* table contains information about the email messages including the message ID (*mid*), the sender of the message (*sender*), the date (*date*), the internal message ID (*message_id*), the subject and body of the message (*subject* and *body*), and the exact folder of the email (*folder*). The *recipientinfo* table contains information about the recipient of a message including the recipient ID (*rid*), the message ID (*mid*), the type of the message (*rtype*), and the email address of the recipient (*rvalue*). The *referenceinfo* table contains messages that have been referenced in other email messages, for example, in a forwarded message; it contains a reference-info ID (*rid*), the message ID (*mid*), and the entire message (*reference*). 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 9.

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 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

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

reflected by the new *forward_msg* relation, which is correspondingly annotated by the *forward_msg* presence condition.

A main focus of Hall's decomposition [24] 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 as described by Hall, such as the *referenceinfo* relation and alternative email addresses.

In addition to providing the schema in the encoding used by our prototype VDBMS tool, we also provide a direct encoding in SQL which generates the universal schema for the VDB. Variation is encoded as an additional relation of the form *vdb_pcs*(*element_id*,*pres_cond*) that captures all of the relevant presence conditions: that of the v-schema itself (i.e. the feature model), and those of each relation and attribute.The *element_id* of the feature model is *variational_schema*; the *element_id* of a relation *r* is its name *r*, and of attribute *a* in relation *r* is *r.a*. The plain SQL encoding of the v-schema supports the use of the case studies for research on the effective management of variation in databases independent of VDBMS.

A.1.3 Populating the Email SPL VDB

The final step to create the email VDB is to populate the database with data from the Enron email dataset, adapted to fit our v-schema [40]. For evaluation purposes, we want the data from the dataset to be distributed across multiple variants of the VDB. To simulate this, we identified five plausible configurations of the email SPL, which we divide the data among. The five considered configurations of the email SPL: *Basic email:* This variant includes only basic email functionality and does not include any of the optional features. *Enhanced email:* This variant extends the basic email system by enabling two of the most commonly used email features: *forwardmessages* and *filtermessages*. *Privacy-focused email:* This variant extends the basic email system with features that focus on privacy. Specifically, the enabled features are: *signature*, *encryption*, and *remailmessage*. *Business email:* This variant extends the basic email system with features tailored to an environment where most emails are expected to be among users within the same business network. Its enabled features are: *addressbook*, *signature*, *encryption*, *autoresponder*, and *mailhost*. *Premium email:* This variant includes all of the optional features in the SPL, that is, all features are enabled. For all variants, any features that are not enabled are disabled.

The original Enron dataset has 150 employees with 252,759 email messages. We load this data into the *employeelist* and *messages* tables defined in Section A.1.2, initializing all attributes that are not present in the original dataset to `NULL`.

For the *employeelist* table, we construct five views corresponding to the five variants of the email system described above. We allocate 30 employees to each view based on their employee ID, that is, the first 30 employees sorted by employee ID are associated with the basic email variant, the next 30 with the enhanced email variant, and so on. The presence condition for each tuple is set to the conjunction of features enabled in that view. We then modify each of the views of the *employeelist* table by adding randomly generated values for attributes associated with the enabled features; e.g., in the view for the privacy-focused variant, we populate the *verification_key* and *public_key* attributes. Any attribute that is not present in the given tuple due to a conflicting presence condition will remain `NULL`. For example, both the *verification_key* and *public_key* attributes remain `NULL` for employees in the enhanced variant view since the presence condition does not include the corresponding features.

For the *messages* table, we again create five views corresponding to each of the variants. Each tuple

Table 10: Evolution of an employee database schema [35].

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*)

is added to the view of the variant that contains the message's sender, which updates the tuple's presence condition accordingly. The *messages* table also contains several additional attributes corresponding to optional features, which we populate in a systematic way. We set *is_signed* to *true* if the message sender has the *signature* feature enabled, and we set *is_encrypted* to *true* if *both* the message sender and recipient have *encryption* enabled. We populate the *is_forward_msg*, *is_autoresponse*, and *is_system_notification* attributes by doing a lightweight analysis of message subjects to determine whether the email is any of these special kinds of messages; for example, if the subject begins with "FWD", we set the *is_forward_msg* attribute to *true*. If a forward or auto-reply message was sent by a user that does not have the corresponding feature enabled, we filter it out of the dataset. After filtering, the *messages* relation contains 99,727 messages. For each forward or auto-reply message, we also add a tuple with the relevant information to the new *forward_msg* and *auto_msg* tables. For employees belonging to database variants that enable *remailmessage*, *autoresponder*, *addressbook*, or *mailhost* we randomly generate tuples in the tables that are specific to each of these features. Finally, the *recipientinfo* relation is imported directly from the dataset. We set each tuple's presence condition to a conjunction of the presence conditions of the sender and recipient.

A.2 Variation in Time: Employee Case Study

In our second case study, we focus on variation that occurs in "time". It demonstrates the use of a VDB to encode an employee database evolution scenario. We systematically adapt an existing database evolution scenario from Moon et al. [35] into a VDB and populate it by a dataset that is widely used in databases research.[6]

A.2.1 Variation Scenario: An Evolving Employee Database

Moon et al. [35] 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 10. In V_1, employees

[6] `https://github.com/datacharmer/test_db`

Table 11: Employee v-schema with feature model m_{emp}.

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

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 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:

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

A.2.2 Generating V-Schema of the Employee VDB

The v-schema for this scenario is given in Table 11. It encodes all five of the schema versions in Table 10 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.

A.2.3 Populating the Employee VDB

Finally, we populate the employee VDB using data from the widely used employee database linked to in this subsection's lede. This database contains information for 240,124 employees. To simulate the evolution of the database over time, we divide the employees into five roughly equal groups based on their hire date within the company. Each group is assumed to have been hired during the lifetime of a particular version of the database, and is therefore added to that version of the database and *also* to all subsequent versions of the

database. This simulates the fact that as a database evolves, older records are typically forward propagated to the new schema [38]. Thus, V_5 contains the records for all $240,124$ employees, while older versions will contain progressively fewer records. The final employee VDB has $954,762$ employee due to this forward propagation, despite having the same number of employees as the original database.

The schema of the employee database used to populate the employee VDB is different from all versions of the v-schema, yet it includes all required information. Thus, we manually mapped data from the original schema onto each version of the v-schema.

B Variational Queries for Variational Case Studies

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 [8].

In this section, we illustrate how variation in software leads to variation in information needs and variation in 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 case study, we provide a set of v-queries. We present only a sample of the queries here for space reasons, yet we provide the full query sets in the GitHub repository linked in Section 1.

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

B.1 Email Query Set

To produce a set of queries for the email SPL case study, we collected all of the information needs that we could identify in the description of the email SPL by Hall [24]. 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. More specifically, the email header includes the sender and receiver of the email, whether an email is signed and the location of a signature verification key, whether an email is encrypted and the location of the corresponding public key, the subject and body of the email, the mail host it belongs to, whether the email should be filtered, and so on. 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.

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

This query extracts the sender, recipient, subject, and body of the email to populate the header. The projection is applied to an intermediate result *mes_rec* constructed by joining the *messages* table with the *recipientinfo* table on recipient IDs; we reuse this intermediate result also in subsequent queries.

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

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

The intermediate result *mes_rec_emp* formed by joining *mes_rec* with the *employeelist* relation will be reused in later queries. 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 [22, 28] to factor commonalities out of choices and reduce redundancy in queries like Q_{bf}.

As another example of a single-feature query, $Q_{forward}$ captures the information needs for implementing the *forwardmessages* feature. It is similar to the previous queries except that it extracts the *forwardaddr* from the *auto_msg* table, which is needed to construct the message header for the new email to be forwarded when email X is received by a user with a *forwardaddr* set.

$$Q_{forward} = \pi_{rvalue,forwardaddr,subject,body} temp$$
$$temp \leftarrow mes\_rec\_emp \bowtie_{employeelist.eid=forward\_msg.eid} auto\_msg$$

The other single-feature queries are similar to those shown here.

Besides single-feature queries, we also provide queries that gather information needed to identify and address the undesirable feature interactions described by Hall [24]. 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.

$$Q_{sf} = signature \wedge forwardmessages\langle \pi_{rvalue,forwardaddr,emp1.is\_signed,emp1.verification\_key} temp, \\ signature\langle Q_{sig}, forwardmessages\langle Q_{forward}, Q_{basic}\rangle\rangle\rangle$$

$$temp \leftarrow ((((\sigma_{mid=X} messages) \bowtie recipientinfo) \bowtie_{sender=emp1.email\_id} (\rho_{emp1} employeelist)) \\ \bowtie_{rvalue=emp2.email\_id} (\rho_{emp2} employeelist)) \bowtie forward\_msg$$

The query Q_{sf} also resolves another consequence of the interaction between these two features. This time Sam successfully verifies message X and forwards it to Sam2 which changes the header in the system s.t. it states message X has been successfully verified, thus, the message could be altered by hackers while it is being forwarded (Hall's interaction 27). The system can use Q_{sf} to generate the correct header in this scenario again.

Some feature interactions require more than one query to satisfy their information need due to VRA's limitation that values cannot be variational. 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.

$$Q_{ef} = encryption \wedge forwardmessages\langle \pi_{rvalue}(\sigma_{mid=X \wedge is\_encrypted} messages), \\ encryption\langle Q_{encrypt}, forwardmessages\langle Q_{forward}, Q_{basic}\rangle\rangle$$

$$Q'_{ef} = encryption \wedge forwardmessages\langle temp, encryption\langle Q_{encrypt}, forwardmessages\langle Q_{forward}, Q_{basic}\rangle\rangle\rangle$$

$$temp \leftarrow \pi_{rvalue,forwardaddr,subject,body}(\sigma_{mid=X \wedge \neg is\_encrypted} \\ (mes\_rec\_emp \bowtie_{employeelist.eid=forward\_msg.eid} forward\_msg))$$

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.

B.2 Employee Query Set

For this case study, we have a set of existing plain queries to start from. Moon et al. [35] 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 A.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:

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

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.