Theory and Implementation of a Variational Database Management System

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

Abstract

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

1 Introduction

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

Consider schema evolution which is an instance of schematic variation in databases that is well-supported [19, 6, 4, 25, 21]. Changes applied to the schema over time are *variation* in the database and every time the database evolves, a new *variant* is generated. Current solutions addressing schema evolution rely on temporal nature of schema evolution by using timestamps [19, 6, 4, 25] or keeping an external file of time-line history of changes applied to the database [21]. These approaches only consider variation in time and do not incorporate the time-based changes into the database directly, rather they *simulate* the effect of these changes, resulting in brittle systems.

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

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

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

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

- Objective 1: Identify the kinds of variation existing in relational databases in different application domains.

- Objective 2: Design a query language and implement a database management system that accommodate variations identified in objective 1.

- Objective 3: Demonstrate how the proposed system can be used to manage variation in databases in different application domains.

- Objective 4: Mechanize proofs of properties of the language and the system.

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

Name	Notation	Section
Feature	f	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 4.2.1
Variational attribute set	A	
Variational relation schema	s	
Variational schema	S	
Variational tuple	u	Section 4.2.2
Variational relation content	U	
Variational table	$t = (s, U)$	
Choice	$e\langle x, y\rangle$	Section 4.2.3
Variational condition	θ	
Variational query	q	

2 Statement of Thesis

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

3 Preliminaries

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

3.1 Relational Databases and Relational Algebra

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

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

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

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

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

instance by *database*.

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

3.2 Formula Choice Calculus and Annotation

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

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

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

Table 2: Objective 1 research questions.

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

4 Research Goals and Methods

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

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

[for all these research questions here explain what you do intuitively and then refer to the formal definition and examples in subsection where they can skip ahead if they want to.]

For RQ1.1 [complete with how variation arises in databases] we explored different application domains where databases change, yet, there is a need to keep and access the older versions (variants) of the database for business reasons. Section 4.1.1 [ref to motivation and add a subsection for dimensions of variation]

For RQ1.2 we introduce a *feature space* that captures dimensions of variation that may appear in a variational database scenario. We then introduce *feature expressions* as propositional formulas of features. Section 4.1.2 explains this encoding in details.

For RQ1.3 ...

4.1.1 Motivating Example

[adjust to proposal and fix refs]

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[1]. 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

[1]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

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 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 $V_1 = \texttt{true}$. This scenario describes *component evolution*: database evolution in SPL resulted from developers update, refactor, improve, and components [11].

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 [11].

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 paper 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*. Section **??** explains how a VDB achieves this. Example 1 shows how v-schema achieves this at schema level for a part of our motivating example and Section **??** discusses two databases that achieve this at both content and schema level.

(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*. Our query language achieves this by introducing variation into queries, Section **??**. Example 2 and Example 3 illustrate this for our motivating example.

(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*. Storing variants that a tuple belongs to in a VDB achieves the first part, Section **??**, and VRA's type system ensures the second part, Section 4.4.2. Example 9 illustrates this for a given 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*. We define *configure* function for a VDB and its elements, Figure 3, in addition to queries, Figure 5, that achieves this. Example 4 illustrate deploying an example query.

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 [11], which still requires a great effort by DBAs and SPL developers and it does not address **N0**-**N2** sufficiently.

4.1.2 Encoding Variability

[fix examples to ones that you have introduced here]

To account for variability in a database we need a way to encode it. To encode variability we first organize the configuration space into a set of features, denoted by **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 $\mathbf{F}$ are used to indicate which parts of a variational entity within the database are different among different variants. Enabling or disabling each of the features in $\mathbf{F}$ produces a particular *variant* of the entity in which all variation has been removed. A *configuration* is a *total* function that maps every feature in the feature set to a boolean value. For brevity, we represent a configuration by the set of enabled features. For example, in our motivating scenario, the configuration $\{V_2, T_3, edu\}$ represents a database variant where only features V_2, T_3, and *edu* are enabled. This database variant contains relation schemas of the employee and education sub-schemas associated with V_2 and T_3 in Table 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 $\{V_2, T_3, edu\}$.

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

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

No matter the context, features often have a relationship with each other that constrains configurations. For example, only one of the temporal features of $V_1 - V_5$ can be $\texttt{true}$ 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 $\mathbb{E}[\![m]\!]_c = \texttt{false}$. 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.

For RQ2.1 we annotate elements of a database with feature expression, as introduced in Section 4.1.2. 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 (Section 4.2.1). At the content level, we annotate each tuple with a feature expression, indicating when the tuple is present (Section 4.2.2).

[add def of vdb and what it is conceptually]

For RQ2.2 ... [ref to v-rel alg]

Feature expression generic object:

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

Feature expression syntax:

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

Relations over feature expressions:

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

Figure 2: Feature expression syntax and relations.

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 go to implement a database management system that uses that framework? (In progress)

For RQ2.3 ... [ref impl]

4.2.1 Variational Schema

[adjust examples accordingly]

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 1. *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`.*

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

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 1 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 1. 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 $\{\underline{empno}, \underline{hiredate}, \underline{title}, \underline{deptno}\}$.

4.2.2 Variational Table

[adjust examples accordingly]

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 [13]. 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.

4.2.3 Variational Relational Algebra

[adjust for proposal]

To account for variation, VRA combines relational algebra (RA) with *choices* [27, 9]. 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.

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 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 2. 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 **??**. In examples, we also use a join operation with a variational condition, $q_1 \bowtie_\theta q_2$, which is syntactic sugar for $\sigma_\theta(q_1 \times q_2)$.

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 Set of Attributes Configuration:

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

V-Relation Schema Configuration:

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

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

V-Schema Configuration:

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

$$\mathbb{S}[\![\{r_1\,(A_1)^{e_1}, \ldots, r_n\,(A_n)^{e_n}\}^m]\!]_c$$

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

V-Tuple Configuration:

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

$$\mathbb{U}[\![(v_1, \ldots, v_l)^{e_u}]\!]_c = (\mathbb{V}[\![v_1]\!]_c, \ldots, \mathbb{V}[\![v_l]\!]_c)$$

$$\textit{where } \forall 1 \leq i \leq l :$$

$$\mathbb{V}[\![v_i]\!]_c = \begin{cases} v_i, & \text{if } \mathbb{E}[\![m \wedge pc(rel(att(v_i))) \wedge pc(att(v_i)) \wedge e_u]\!]_c \\ \varepsilon, & \text{otherwise} \end{cases}$$

V-Relation Content Configuration:

$$\mathbb{T}[\![.]\!] : \mathbf{T} \to \mathbf{C} \to \underline{\mathbf{T}}$$

$$\mathbb{T}[\![\{u_1, \ldots, u_k\}]\!]_c = \{\mathbb{U}[\![u_1]\!]_c, \ldots, \mathbb{U}[\![u_k]\!]_c\}$$

VDB Instance Configuration:

$$\mathbb{I}[\![.]\!] : \mathbf{I} \to \mathbf{C} \to \mathbf{I}$$

Variational conditions:

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

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. [remember that you removed join (also removed it from query config def and constrain query by schema). if you want use it just say it's a syntactic sugar.]

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 paper.

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 $\mathit{result}({a_1}^{f_2})^{f_3}$, 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., $\mathit{unsat}(f_3 \wedge \neg f_3)$). 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 2. *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:*

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

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.

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

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 6.

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 3. *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.*

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

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.

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

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 4. *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: $\mathbb{Q}[\![q_5]\!]_{\{f_1\}} = \mathbb{Q}[\![q_5]\!]_{\{\ \}} = \pi_{\underline{a}_1} \underline{r}$, $\mathbb{Q}[\![q_5]\!]_{\{f_2\}} = \pi_{\underline{a}_1, \underline{a}_3} \underline{r}$, $\mathbb{Q}[\![q_5]\!]_{\{f_1, f_2\}} = \pi_{\underline{a}_1, \underline{a}_2, \underline{a}_3} \underline{r}$.*

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

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 $\underline{\varepsilon}$.

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 5. *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 $q =$
$(f_1 \wedge \neg f_2 \wedge \ldots \wedge \neg f_n)\langle \underline{q}_1, (f_2 \wedge \ldots \wedge \neg f_n)\langle \underline{q}_2, \ldots f_n\langle \underline{q}_n, \varepsilon\rangle \ldots\rangle\rangle$. $\square$

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.

4.2.4 VDBMS Implementation

[adjust for proposal]

[the annotation of tuples is encoded as an attribute in implementation.] To interact with VDBs using v-queries, we implement *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 v-tables, explained in Section 4.2.2. 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.

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? What are the steps to generate a VDB from a scenario of variation in a database? (VaMoS'21)
RQ3.2: What are the benefits and drawbacks of representing variation generically in databases as opposed to having scenario-tailored approaches? (VaMoS'21)

VDBMS architecture: Figure 6 shows the architecture of VDBMS and its modules. For now, we assume a VDB and its v-schema are generated by an expert and are stored in a DBMS, we return to generation of VDBs in Section **??**. 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 constrained by 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, Section **??** provides three approaches for this. Example **??** in Appendix **??** demonstrates the flow of a v-query through VDBMS.

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 can be used to manage variation in databases in different application domains

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

For RQ3.1 ...

For RQ3.2 ...

4.4 Mechanize proofs of properties of the language and the system

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

For RQ4.1 ... [ref to vdb prop]

For RQ4.2 .. [don't forget refs, intuitive explanation, formal def ref, ex ref]

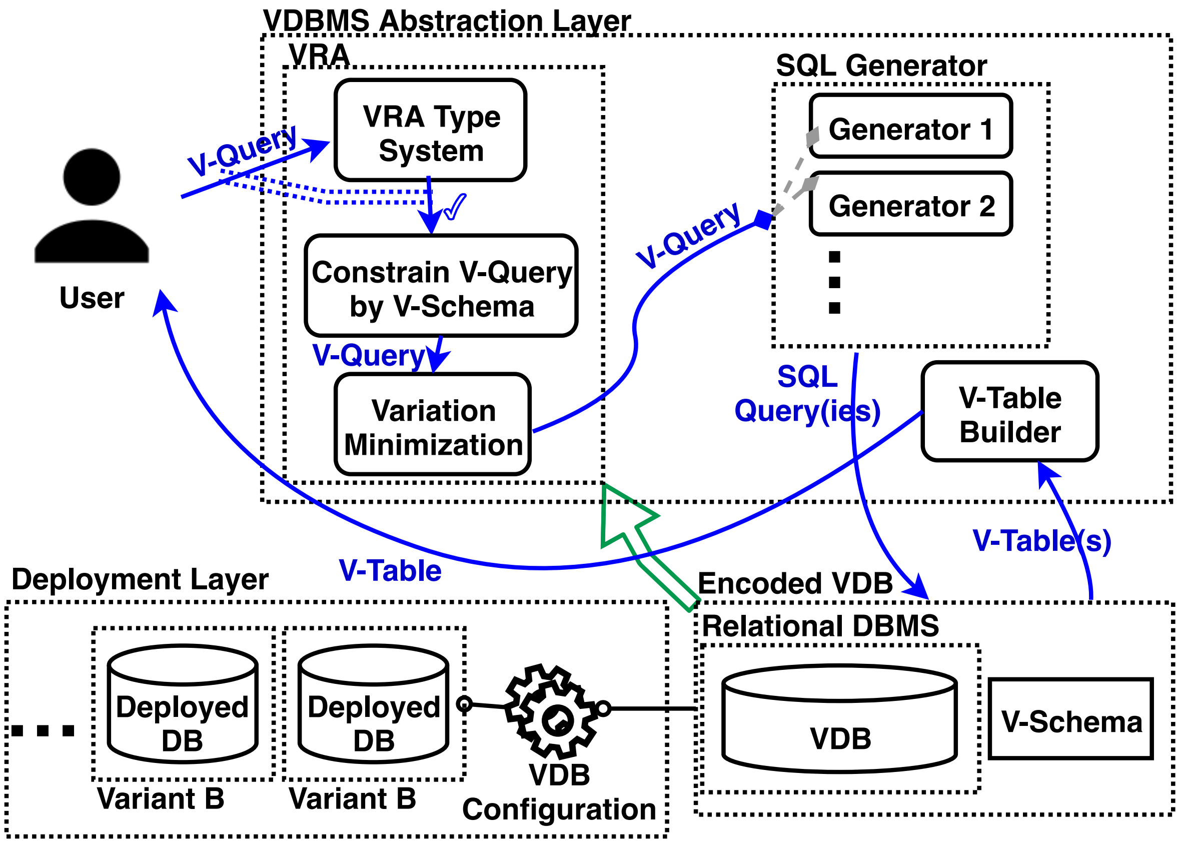

Figure 6: VDBMS architecture and execution flow of a v-query. The dotted double-line from v-query to pushing v-schema module indicates the dependency of passing the v-query to this module only if it is valid. The dashed gray arrows with diamond heads demonstrate an option for the flow of input. The blue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

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 RQ4.3 ...

4.4.1 Property Checking over a VDB

[adjust for proposal]

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 this paper satisfy all of them. Depending on the context of the VDB, more specialized properties can be checked too. For example, if temporal variability in a database is accumulated over variants, i.e., 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

[adjust for proposal]

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 4 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 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 **??**. 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 **??**, 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 **??**, filters both attributes and their presence conditions. Example 6 illustrates generating the type of a query step by step.

Example 6. *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 2*[2]*. 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*

[2]Derivation trees of these examples can be fine here! [Eric, do we need them? If yes, where should we put them?]

V-queries typing rules:

EMPTYRELATION-E

$$e,S \vdash \varepsilon : \{\ \}^{\texttt{false}}$$

RELATION-E

$$\frac{r(A)^{e'} \in S \qquad sat\big(e \wedge e'\big)}{e,S \vdash r : A^{e \wedge e'}}$$

PROJECT-E

$$\frac{e,S \vdash q : A'^{e'} \qquad |\downarrow(A^{e})| = |A| \qquad A \prec \downarrow\left(A'^{e'}\right)}{e,S \vdash \pi_A q : \left(A \cap A'\right)^{e'}}$$

SELECT-E

$$\frac{e,S \vdash q : A^{e'} \qquad e,\downarrow\left(A^{e'}\right) \vdash \theta}{e,S \vdash \sigma_\theta q : A^{e'}}$$

CHOICE-E

$$\frac{e \wedge e', S \vdash q_1 : A_1^{e_1} \qquad e \wedge \neg e', S \vdash q_2 : A_2^{e_2}}{e,S \vdash e'\langle q_1, q_2\rangle : \left(\downarrow\left(A_1^{e_1}\right) \cup \downarrow\left(A_2^{e_2}\right)\right)^{(e_1 \wedge e') \vee (e_2 \wedge \neg e')}}$$

PRODUCT-E

$$\frac{e,S \vdash q_1 : A_1^{e_1} \qquad e,S \vdash q_2 : A_2^{e_2} \qquad \downarrow\left(A_1^{e_1}\right) \cap \downarrow\left(A_2^{e_2}\right) = \{\}}{e,S \vdash q_1 \times q_2 : \left(\downarrow\left(A_1^{e_1}\right) \cup \downarrow\left(A_2^{e_2}\right)\right)^{e_1 \wedge e_2}}$$

SETOP-E

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

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

CONJUNCTION-C

$$\frac{e,A \vdash \theta_1 \qquad e,A \vdash \theta_2}{e,A \vdash \theta_1 \wedge \theta_2}$$

DISJUNCTION-C

$$\frac{e,A \vdash \theta_1 \qquad e,A \vdash \theta_2}{e,A \vdash \theta_1 \vee \theta_2}$$

CHOICE-C

$$\frac{e \wedge e', A \vdash \theta_1 \qquad e \wedge \neg e', A \vdash \theta_2}{e,A \vdash e'\langle \theta_1, \theta_2\rangle}$$

NEG-C

$$\frac{e,A \vdash \theta}{e,A \vdash \neg\theta}$$

ATTOPTVAL-C

$$\frac{a^{e'} \in A \qquad sat\big(e' \wedge e\big) \qquad k \in dom_{\mathscr{I}}(a)}{e,A \vdash \underline{a} \bullet k}$$

BOOLEAN-C

$$e,A \vdash b$$

ATTOPTATT-C

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

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.

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 $A_{right} = \{\ \}^{\mathtt{false}}$, 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 **??**, 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 **??**, for v-sets of attributes.

4.4.3 Explicitly Annotating Queries

[adjust for proposal]

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 2 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:

$$q_5 = \pi_{empno^{V_4 \vee V_5}, name^{V_4}, firstname^{V_5}, lastname^{V_5}} m_2\langle empbio, \varepsilon\rangle \quad .$$

This is the unsimplified version:

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

Similarly, the outer projection in the query q_6 written over S_2

$$q_6 = \pi_{name, firstname} V_4\langle \pi_{name} q_5, \pi_{firstname} q_5\rangle$$

does not repeat the presence conditions of attributes from its subquery's type. The query

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

Figure 8: Constraining queries by the underlying v-schema. Queries passed to this function are well-typed.

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

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 $\mathbf{Q} \rightarrow \mathbf{S} \rightarrow \mathbf{Q}$, 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.

Theorem 7. *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[3].

We illustrate the application of Theorem 7 for queries q_1 and q_5. Example 6 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 w.r.t. Schema

[adjust for proposal]

Similar to other applications of variational research [**?** **?**], the type system must preserve the variation encoded in a v-query. We define the *variation-preserving property w.r.t. 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 8 defines this property formally.

[3]We proved this theorem in the Coq proof assistant. The encoding of the theorem and the proof can be fine here: [update later. you may have to upload extra files instead of providing a link if submission is double-blind.]

$$\begin{tikzcd}
\lfloor q \rfloor_S \arrow[r, "type"] \arrow[d, dashed, "\mathbb{Q}[\![\cdot]\!]_c"'] & A^e \arrow[d, dashed, "\mathbb{A}[\![\cdot]\!]_c"] \\
\underline{q} \arrow[r, "\underline{type}"] & \underline{A}
\end{tikzcd}$$

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 and RA's type safety [**?**] implies that VRA's type system is also type safe. Example 9 illustrates why the query must be constrained by the v-schema in the variation-preserving diagram.

Theorem 8. *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.,*

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

Proof. We proved this theorem in the Coq proof assistant. [It can be find here.] [Eric, maybe we have a sketch of it if we have space?] □

Example 9. *Consider the v-query $q_5 = \pi_{a_1, {a_2}^{f_1 \wedge f_2}, {a_3}^{f_2}} r$ given in Example 4. 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 constrained query by 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

[adjust for proposal]

[Eric, I kept this here and I just point out this property of VRA in Section 4.2.3 in a note box (could you please review that too?). How do you feel about moving this subsection to appendix?] VRA is flexible since an information need can be represented via multiple v-queries as demonstrated in Example 2 and Example 3. 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 4.6 summarizes the connections between the research questions and activities. Table 4.6 provides the timeline for this proposal.

5 Related Work

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

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

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	Early 2021	Complete
2.2	Poly'18, DBPL'17, VLDB'21	Early 2021	Complete
2.3	VLDB'21	Early 2021	In progress
3.1	VaMoS'21	N/A	Complete
3.2	VaMoS'21	N/A	Complete
4.1	VaMoS'21	N/A	Complete
4.2	VLDB'21, TOPLAS'21	Mid 2021	In progress
4.3	TOPLAS'21	Mid 2021	Not started

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

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

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

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

Variational research:

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

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

Database versioning: As mentioned in Section 4.2.2, database versioning approaches only consider content-level variation [13] which is usually used for experimental and scientific databases.

6 Conclusion

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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