AN ABSTRACT OF THE DISSERTATION OF

Parisa Ataei for the degree of Doctor of Philosophy in Computer Science presented on June 20, 2021.

Title: Theory and Implementation of a Variational Database Management System

Abstract approved: _____

Eric Walkingshaw

In this thesis I present the variational database management system, a formal framework and its implementation for representing variation in relational databases and managing variational information needs. A variational database is intended to support any kind of variation in a database. Specific kinds of variation in databases have already been studied and are well-supported, for example, schema evolution systems address the variation of a database's schema over time and data integration systems address variation caused by accessing data from multiple data sources simultaneously. However, many other kinds of variation in databases arise in practice, and different kinds of variation often interact, but these scenarios are not well-supported by the existing work. For example, neither the schema evolution systems nor the database integration systems can address variation that arises when data sources combined in one database evolve over time.

This thesis collects a large amount of work: It defines the variational database framework and the syntax and [specific kind of] semantics of the variational relational algebra, a query language for variational databases. It also defines the requirements of a generic variational database framework that makes the framework expressive enough to encode any kind of variation in databases. Additionally, it [shows/proves] that the introduced framework satisfies all these needs. It presents two use cases of the variational database framework that are based on existing data sets and scenarios that are partially supported by existing techniques. It presents the variational database management system which is the implementation of variational databases and variational relational algebra as an abstract layer written in Haskell on top of a traditional RDBMS. It also presents several

theoretical results related to the framework and query language, such as syntax-based equivalence rules that preserve the semantics of a query, a type system for ensuring that a variational query is well formed with respect to the underlying variational schema, and a confluence property of the variational relational algebra type system and semantics with respect to the relational algebra type system and semantics.

Theory and Implementation of a Variational Database Management System

by

Parisa Ataei

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 20, 2021
Commencement June 2021

Doctor of Philosophy dissertation of Parisa Ataei presented on June 20, 2021.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Parisa Ataei, Author

ACKNOWLEDGEMENTS

[Eric. Committee. jeff. abu. parents. friends.]

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

LIST OF ALGORITHMS

Algorithm Page

Chapter 1: Introduction

Managing variation in databases is a perennial problem in database literature and appears in different forms and contexts [27, 6, 11, 16, 8] and it is inevitable [26]. Variation in databases mainly arises when multiple database instances conceptually represent the same database, but, differ in their schema, content, or constraints. Specific kinds of variation in databases have been addressed by context-specific solutions, such as schema evolution [21, 10, 5, 25, 22], data integration [13], and database versioning [9, 19]. However, there are no generic solution that addresses any kind of variation in databases. We motivate the need for a generic solution to variation in databases in Section 1.1.

The major contribution of this thesis is the *variational database* framework, a generic relational database framework that explicitly accounts for database changes (variation), and *variational relational algebra*, a data manipulation language for our framework that allows for information extraction from a variational database. The variational framework is generic because it can encode any kind of variation in databases. Additionally and more importantly, it is designed such that it satisfies any information need that a user may have in a variational database scenario. Based on information needs in a variational database scenario, we define requirements of a variational database framework in Section 1.1 and throughout the thesis, we show that our framework satisfies these requirements.

In addition to a formal description of the variational database framework and variational relational algebra and some theoretical results, this thesis distributes and presents two variational data sets (including both a variational database and a set of queries) as well as a *variational database management system* that implements the variational database framework as an abstraction layer on top of a traditional relational database management system in Haskell. Section 1.2 enumerates the specific contributions in the contest of an outline of the structure of the rest of the thesis.

1.1 Motivation and Impact

Managing variation in databases is a perennial problem in database literature and appears in different forms and contexts [27, 6, 11, 16, 8]. Variation in databases mainly arises when multiple database instances conceptually represent the same database, but, differ in their schema or content. The variation in schema and/or content occurs in two *dimensions*: time and space. Variation in space refers to different variants of database that coexist in parallel while variation in time refers to the evolution of database, similar to variation observed in software [28]. Note that variation in a database can occur due to both dimensions at the same time.

Existing work on variation in databases addresses specific kinds of variation in a context and proposes solutions specific to the context of the variation, such as schema evolution [21, 10, 5, 25, 22], data integration [13], and database versioning [9, 19]. Unfortunately, some of these tools do not address all their user's needs. Furthermore, they are all *variation-unaware*, i.e., they dismiss that they are addressing a specific kind of a bigger problem. Thus, not only they cannot address a new kind of database variation but they also cannot address the interaction of different kinds of database variation since they assume that each kind of variation is *isolated* from another kinds. This costs database researchers to develop a new system for every individual kind of data variation and forces developers and DBAs to use manually unsafe workaround.

For example, schema evolution is a kind of schematic variation in databases over time that is well-supported [21, 10, 5, 25, 22]. 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 dismiss that it is a kind of variation, thus, they *simulate* the effect of variation by relying on temporal nature of schema evolution and using timestamps [21, 10, 5, 25] or keeping an external file of time-line history of changes applied to the database [22].

Unlike schema evolution, some kinds of database variation are partially supported. For example, while developing software for multiple clients simultaneously, an approach called software product line (SPL) [12], a different database schema is required for each client due to client's different business requirements and environments [24]. SPL researchers have developed encodings of data models that allow for arbitrary variation by annotating different elements of the model with features from the SPL [24, 23, 2]. Thus,

they can generate a database schema *variant* for each software *variant* requested by a client and generated by the SPL. However, these solutions address *only* variation in the data model but do not extend it to the level of the data or queries. The lack of variation support in queries leads to unsafe techniques such as encoding different variants of query through string munging, while the lack of variation support in data precludes testing with multiple variants of a database at once.

The situation exacerbates even more when two kinds of variation interact and create a new kind of database variation: the evolution of the database used in development of software by an SPL. This is due to the evolution of software and its artifacts; an inevitable phenomena [18]. This is where even previous context-specific solutions like schema evolution tools fail because the context-specific approaches assume that the kind of variation that they address is completely isolated from other kinds of variation. We motivate this case through an example in Section 1.1. Note that new kinds of variation may arise that are not necessarily the interaction of already-addressed variation in database. We discuss such a scenario in Section 1.1.2.

As we have shown, variation in databases is abundant and inexorable [26]; impacts DBAs, data analysts, and developers significantly [18]; and appears in different contexts. Current methods are all extremely tailored to a specific context. Consequently, they fail to address the interaction of their specific kind of database variation with other kinds. Hence, the challenge becomes defining a variation-aware database that can can model different kinds of variation in different contexts such that it satisfies different specialists' needs at different stages, e.g., development, information extraction, deployment, and testing.

1.1.1 Motivating Example

In this section, we motivate the interaction of two kinds of variation in databases resulting in a new kind: database-backed software produced by an SPL and schema evolution. An SPL uses a set of boolean variables called *features* to indicate the functionalities that each software variant requires. Consider an SPL that generates management software for companies. It has a feature *edu* that indicates whether a company provides educational means such as courses for its employees. Software variants in which *edu* is disabled (i.e., *edu* = `false`) only provide basic functionalities while ones in which *edu* is enabled

Table 1.1: Schema variants of the employee database developed for multiple software variants by an SPL. Note that an educational database variant must contain a basic database variant too.

(a) Database schema variants for basic software variants.

Temporal Features	basic **Database Schema Variants**
V_1	*engineerpersonnel* (*empno*, *name*, *hiredate*, *title*, *deptname*) *otherpersonnel* (*empno*, *name*, *hiredate*, *title*, *deptname*) *job* (*title*, *salary*)
V_2	*empacct* (*empno*, *name*, *hiredate*, *title*, *deptname*) *job* (*title*, *salary*)
V_3	*empacct* (*empno*, *name*, *hiredate*, *title*, *deptno*) *job* (*title*, *salary*) *dept* (*deptname*, *deptno*, *managerno*) *empbio* (*empno*, *sex*, *birthdate*)
V_4	*empacct* (*empno*, *hiredate*, *title*, *deptno*, *std*, *instr*) *job* (*title*, *salary*) *dept* (*deptname*, *deptno*, *managerno*) *empbio* (*empno*, *sex*, *birthdate*, *name*)
V_5	*empacct* (*empno*, *hiredate*, *title*, *deptno*, *std*, *instr*, *salary*) *dept* (*deptname*, *deptno*, *managerno*, *stdnum*, *instrnum*) *empbio* (*empno*, *sex*, *birthdate*, *firstname*, *lastname*)

(b) Database schema variants for educational software variants.

Temporal Feature	educational **Database Schema Variants**
T_1	*course* (*coursename*, *teacherno*) *student* (*studentno*, *coursename*)
T_2	*course* (*courseno*, *coursename*, *teacherno*) *student* (*studentno*, *courseno*)
T_3	*course* (*courseno*, *coursename*) *teach* (*teacherno*, *courseno*) *student* (*studentno*, *courseno*, *grade*)
T_4	*ecourse* (*courseno*, *coursename*) *course* (*courseno*, *coursename*, *time*, *class*) *teach* (*teacherno*, *courseno*) *student* (*studentno*, *courseno*, *grade*)
T_5	*ecourse* (*courseno*, *coursename*, *deptno*) *course* (*courseno*, *coursename*, *time*, *class*, *deptno*) *teach* (*teacherno*, *courseno*) *take* (*studentno*, *courseno*, *grade*)

provide educational functionalities in addition to the basic ones. Thus, this SPL yields two types of variants: basic and educational.

Each variant of this SPL needs a database to store information about employees, but SPL features impact the database: While basic variants do not need to store any education-related records educational variants do. We visualize the impact of features on the schema variants in Table 1.1: It has two schema types: basic, shown in Table 1.1a, and educational, shown in Table 1.1b. A basic schema variant contains only the schema in one of the cells in Table 1.1a while an educational schema variant consists of two sub-schemas: one from the basic and another from Table 1.1b. For example, the yellow highlighted cells include relation schemas for an educational schema variant.

Rows of Table 1.1 indicate the evolution of schema variants in time. To capture the evolution of the software and its database we add two disjoint sets of features which again are boolean variables. The temporal feature sets are disjoint to allow individual paces for the evolution of each type of schema. For example, yellow cells of Table 1.1 show a valid schema variant even though the basic and educational sub-schemas are not in the same row.

Now, consider the following scenario: In the initial design of the basic database, SPL DBAs settle on three tables *engineerpersonnel*, *otherpersonnel*, and *job*; shown in Table 1.1a and associated with 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 feature V_2. Since some clients' software relies on a previous design the two schemas have to coexist in parallel. Therefore, the existence (presence) of *engineerpersonnel* and *otherpersonnel* relations is *variational*, i.e., they only exist in the basic schema when $V_1 =$ `true`. This scenario describes *component evolution*: developers update, refactor, and improve components including the database [18].

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 specific subjects. Hence, an SPL developer needs to enable the *edu* feature for this client, forcing the adjustment of the schema variant to educational. This case describes *product evolution*: database evolution in SPL resulted from clients adding/removing features/-components [18].

The situation is further complicated since the basic and educational schemas are in-

terdependent: Consider the basic schema variant for feature V_4. Attributes *std* and *instr* only exists in the *empacct* relation when *edu* = `true`, represented by a dash-underline, 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 example demonstrates how different kinds of variation interact with each other, an indispensable consequence of modern software development. The described interaction is similar to a recent scenario we discussed with an industry contact in Section 1.1.2.

1.1.2 New Instance of Data Variation in Industry

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

1.1.3 Requirements of a Variation-Aware Database Framework

For a variation-aware framework to be expressive enough to encode any kind of variation in databases, it must satisfy some requirements. Thus, we define the requirements that make a database framework variational through studying different kinds of variation in databases. A variational artifact, including databases, encompasses multiple *variants* of the artifact and provides a way to distinguishes between different variants that are all encoded in one place, the variational artifact. It also provides a way to get the variants from the variational artifact, we call this *configuration function*. These requirements that help distinguish a database framework that simulates the effect of variation compared to one that is variational are listed below:

(R0) *All database variants must be accessible at a given time.* For example, in our motivating example, a company that started with V_1 of the basic schema evolves over time but its different branches adopt the new schema at different paces, thus, it requires access to all variants of the basic schema.

(R1) *The query language must allow for querying multiple database variants simultaneously. Additionally, it must allow for filtering tuples to specific variants.* That is, the framework must provide a query language that allows users to query multiple database variants at the same time in addition to giving them the freedom to choose the variants that they want to query. For example, an SPL tester that is testing a piece of code for the not highlighted variants of the software in Table 1.1 needs to write queries that exclude the variants associated with yellow cells of Table 1.1.

(R2) *Every piece of data must clearly state the variant it belongs to and this information must be kept throughout the entire framework.* Continuing the example of the SPL tester, they need to know the variant that some results belong to in order to be able to debug the software correctly and accordingly.

(R3) *The variational database must provide a way for generating database and query variants.* For example, the SPL developers need to deploy the management software for each client, thus, they need to configure the database schema and its queries in the code for each software variant.

Throughout the thesis, we show how the proposed variational database framework satisfies these requirements via examples, proofs, and tests.

1.2 Contributions and Outline of this Thesis

[to be filled out when I have the chapters.]

The high level goal of this thesis is to emphasize the need for a variation-aware database framework and to present one such framework. Therefore, in addition to the formal definition of the framework and query language, it also provides variational data sets (including both the variational database and a set of queries) to illustrate the feasibility of the proposed framework. Furthermore, it illustrates various approaches to implement such a framework and compares their performance.

The rest of this section describes the structure of this thesis, enumerating the specific contribution that each chapter makes.

Chapter 2 (*Background*) introduces several concepts and terms that are the basis of this thesis. It describes types and how to interpret them. It explains relational databases with assumptions that are held throughout the thesis and relational algebra. It also describes various ways of incorporating variation into elements of a database.

[Chapter 3]

[Chapter 4]

[Chapter 5]

[Chapter 6]

Chapter 7 (*Related Work*) collects research related to different kinds of variation in databases and other related variational research.

Finally, Chapter 8 (*Conclusion*) briefly presents ...

Chapter 2: Background

The core of this thesis is injecting a new aspect to relational databases: *variation*. Thus, the goals of this chapter are twofold: first, to introduce how variation is encoded and represented in our variational database framework; second, to provide the reader with the concepts and notations used to build up the main contributions of this thesis, mainly relational databases and relational algebra.

Section 2.1 describes the database model of relational databases and the specification of the structured used to store the data. Section 2.2 describes SPJC relational algebra to query relational databases. Then, Section 2.3 defines our choice of encoding for the variation space used in the variational database as propositional formulas of boolean variables. Finally, we introduce main approaches used to incorporate variation into our variational database framework. Section 2.4 introduces variation into sets which forms the basis of the variational database framework and Section 2.5 describes the formula choice calculus used to incorporate variation into relational algebra.

2.1 Relational Databases

[add examples of tables.] [add figure of all definitions]

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 $\underline{a}_i \in \underline{\mathbf{A}}$ is an *attribute* contained in a relation named r. Note that there is a total order $\leq_{\underline{\mathbf{A}}}$ on $\underline{\mathbf{A}}$ and every listed set of attributes is written according to $\leq_{\underline{\mathbf{A}}}$. For theoretical development, it suffices to use the same domain of values for all of the attributes. Thus, we fix a countably infinite set *domain dom*. A *value* and a *constant* are elements of *dom*.

The content of database $\underline{D}$ is stored in the form of *tuples*. A tuple $\underline{u}$ is a mapping between an ordered set of 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 $\{\underline{u}_1, \ldots, \underline{u}_m\}$. $att(i)$ returns the attribute corresponding to index i in a tuple. A *table* $\underline{t} = (\underline{s}, \underline{U})$ is a pair of

relation schema and relation content. A *database instance*, $\underline{\mathcal{I}}$, of the database $\underline{D}$ with the schema $\underline{S}$, is a set of tables $\{\underline{t}_1, \ldots, \underline{t}_n\}$. For brevity, when it is clear from the context we refer to a database instance by *database*.

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*, $\underline{\mathcal{I}}$, of the database $\underline{D}$ with the schema $\underline{S}$, is a set of tables $\{\underline{t}_1, \ldots, \underline{t}_n\}$. For brevity, when it is clear from the context we refer to a database instance by *database*.

2.2 The SPJR Relational Algebra

[relational algebra] [add syntax definition] [add type system] [add examples with tables] [maybe add semantics later on]

We do not extend the notation of using underline for relational algebra operations. Instead, relational algebra operations are overloaded and they are used as both plain relational and variational operations. It is clear from the context when an operation is variational or not. We also extend relational algebra s.t. projection of an empty list from a query is valid and it returns an empty set. In fact, we denote such query by the *empty* query ε, thus, $\varepsilon = \pi_{\{\}}\underline{q}$.

[the following is from prelim. revise.]

[add bullet and conditions and attribute list to the definition.]

Figure 2.1 defines the syntax of relational algebra which allows users to query a relational database [1]. The first five constructs are adapted from relational algebra: A query may simply *reference* a relation $\underline{r}$ in the schema. *Renaming* allows giving a name to an intermediate query to be referenced later. Note that $\underline{r}$ is an overloaded symbol that indicates both a relation and a relation name. A *projection* enables selecting a

$$\underline{\theta} \in \underline{\mathbf{\Theta}} \;\coloneqq\; \texttt{true} \mid \texttt{false} \mid a \bullet k \mid a \bullet a \mid \neg\underline{\theta} \mid \underline{\theta} \vee \underline{\theta}$$

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

Figure 2.1: Syntax of relational algebra.

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

2.3 Variation Space in a Variational Database Framework

[have to revise] [define oplus in fig as syntactic sugar.]

To account for variability in a database we need to encode it. To encode variability we introduce a *feature space* $\mathbf{F}$ as a closed set of boolean variables. A feature $f \in \mathbf{F}$

can be enabled (i.e., f= `true`) or disabled (f= `false`). Features capture the variation in a given variational scenario. 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 1.1); for SPLs, the features can be adopted from the SPL feature set (e.g. the *edu* feature in our motivating example, Table 1.1); and for data integration, the features can be representatives of resources.

Features describe local points of variation in a database. Thus, enabling or disabling all the features produces a particular database *variant* where all variation has been removed. A *configuration* is a *total* function that maps every feature in the feature set to a boolean value. 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 in the yellow cells of Table 1.1. We refer to a variant with configuration c as variant c, e.g., 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 2.2. The evaluation function of feature expressions $\mathbb{E}[\![e]\!]_c : \mathbf{E} \to \mathbf{C} \to \mathbf{B}$ simply substitutes each feature f in the expression e with its boolean value set in the given configuration c and then simplifies the propositional formula to a boolean value. For example, assuming that $\mathbf{F} = \{f_1, f_2\}$ $\mathbb{E}[\![f_1 \vee f_2]\!]_{\{f_1\}} =$ `true` $\vee$ `false` $=$ `true`, however, $\mathbb{E}[\![f_1 \vee f_2]\!]_{\{\}} =$ `false` $\vee$ `false` $=$ `false`. Additionally, we define the binary *equivalence* ($\equiv$) relation and the unary *satisfiable (sat)* and *unsatisfiable (unsat)* relations over feature expressions in Figure 2.2.

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. $pc(x^e)$ returns the presence

Feature expression syntax:

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

Relations over feature expressions:

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

Figure 2.2: Feature expression syntax and relations.

condition of the variational element x^e. For example, the annotated number $2^{f_1 \vee f_2}$ is present in variants $\{f_1\}$, $\{f_2\}$, $\{f_1, f_2\}$ but it does not exist in variant $\{\}$. Here, $pc\big(2^{f_1 \vee f_2}\big) = f_1 \vee f_2$.

No matter the context, features often have a relationship with each other that constrains configurations. For example, only one of the temporal features of $V_1 - V_5$ can be `true` for a given variant. This relationship is 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 it follows from the $\mathbb{E}[\![.]\!]_c$ definition that $\mathbb{E}[\![m]\!]_c$= `false`. For example, the restriction that at a given time only one of temporal features $V_1 - V_5$ can be enabled is represented by: $V_1 \oplus V_2 \oplus V_3 \oplus V_4 \oplus V_5$, where $f_1 \oplus f_2 \oplus \ldots \oplus f_n$ is syntactic sugar for $(f_1 \wedge \neg f_2 \wedge \ldots \wedge \neg f_n) \vee (\neg f_1 \wedge f_2 \wedge \ldots \wedge \neg f_n) \vee (\neg f_1 \wedge \neg f_2 \wedge \ldots \wedge f_n)$, i.e., features are mutually exclusive.

2.4 Variational Set

[havve to revise this]

A *variational set (v-set)* $X = \{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}$ is a set of annotated elements [15, 30, 7]. We typically omit the presence condition `true` in a variational set, e.g., $4^{\texttt{true}} = 4$. Conceptually, a *variational set* represents many different plain sets simultaneously. Thus, a plain set $\underline{X} \in \underline{\mathcal{X}}$ is a *variant* of the variational set $X \in \mathcal{X}$ and given variant $\underline{X}$'s

Semantics of feature expressions:

$$
\begin{aligned}
\mathbb{E}[\![.]\!] &: \mathbf{E} \to \mathbf{C} \to \mathbf{B} \\
\mathbb{E}[\![b]\!]_c &= b \\
\mathbb{E}[\![f]\!]_c &= c\ f \\
\mathbb{E}[\![\neg f]\!]_c &= \neg \mathbb{E}[\![f]\!]_c \\
\mathbb{E}[\![e_1 \wedge e_2]\!]_c &= \mathbb{E}[\![e_1]\!]_c \wedge \mathbb{E}[\![e_2]\!]_c \\
\mathbb{E}[\![e_1 \vee e_2]\!]_c &= \mathbb{E}[\![e_1]\!]_c \vee \mathbb{E}[\![e_2]\!]_c
\end{aligned}
$$

Figure 2.3: Feature expression evaluation and functions.

configuration $c \in \mathbf{C}$ it can be generated by evaluating the presence condition of each element with c and including the element if the said evaluation results in `true` and excluding it otherwise. This is done by the *v-set configuration* function $\mathbb{X}[\![X]\!]_c : \mathcal{X} \to \mathbf{C} \to \underline{\mathcal{X}}$. For example, assume we have the feature space $\mathbf{F} = \{f_1, f_2\}$ and the v-set $X_1 = \{2^{f_1}, 3^{f_2}, 4\}$. X_1 represents four plain sets: $\{2,3,4\} = \mathbb{X}[\![X_1]\!]_{\{f_1,f_2\}}$, i.e., the variant $\{2,3,4\}$ is generated from the v-set X_1 under the configuration $f_1 = \texttt{true}, f_2 = \texttt{true}$; $\{2,4\} = \mathbb{X}[\![X_1]\!]_{\{f_1\}}$; $\{3,4\} = \mathbb{X}[\![X_1]\!]_{\{f_2\}}$; and $\{4\} = \mathbb{X}[\![X_1]\!]_{\{\}}$. Following database notational conventions we drop the brackets of a variational set when used in database schema definitions and queries.

A variational set itself can also be annotated with a feature expression. $X^e = \{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}^e$ is an *annotated v-set*. Annotating a v-set with the feature expression e restricts the condition under which its elements are present, i.e., it forces elements' presence conditions to be more specific. The *normalization* operation $\downarrow(X^e)$ applies this restriction: $\downarrow(X^e) = \{{x_i}^{e_i \wedge e} \mid x_i^{e_i} \in X^e, sat(e_i \wedge e)\}$. Note that the *normalization* operation also removes the elements with unsatisfiable presence conditions and it can also be applied to a v-set X since $X^{\texttt{true}} = X$. For example, the annotated v-set $X_1 = \{2^{f_1}, 3^{\neg f_2}, 4, 5^{f_3}\}^{f_1 \wedge f_2}$ indicates that all the elements of the set can only exist when both f_1 and f_2 are enabled. Thus, normalizing the v-set X_1 results in $\{2^{f_1 \wedge f_2}, 4^{f_1 \wedge f_2}, 5^{f_1 \wedge f_2 \wedge f_3}\}$. The element 3 is dropped since $\neg sat(pc(3, X_1))$, where $pc(3, X_1) = \neg f_2 \wedge (f_1 \wedge f_2)$. Note that we use the function $pc(x, X^e)$ to return the presence condition of a unique variational element within a bigger variational structure.

We provide some operations over v-sets used mainly in the Section **??** and defined formally in Appendix **??**. Intuitively, these operations should behave such that configuring the result of applying a variational set operation should be equivalent to applying the plain set operation on the configured input v-sets for all valid configurations. Thus, for all possible operations, $\odot$, defined on v-sets the property $(P_1) : \forall c \in \mathbf{C}.\mathbb{X}[\![\downarrow(X_1) \odot \downarrow (X_2)]\!]_c = \mathbb{X}[\![X_1]\!]_c \underline{\odot} \mathbb{X}[\![X_2]\!]_c$ must hold, where $\underline{\odot}$ is the counterpart operation on plain sets.

Definition 2.4.1 (V-set union) *The* union *of two v-sets is the union of their elements with the disjunction of presence conditions if an element exists in both v-sets:* $X_1 \cup X_2 = \{x^{e_1} \mid x^{e_1} \in\downarrow (X_1), \exists e_2.x^{e_2} \notin\downarrow (X_2)\} \cup \{x^{e_2} \mid x^{e_2} \in\downarrow (X_2), \exists e_1.x^{e_1} \notin\downarrow (X_1)\} \cup \{x^{e_1 \vee e_2} \mid x^{e_1} \in\downarrow (X_1), x^{e_2} \in\downarrow (X_2)\}$. *For example, the union of two normalized v-sets* $\{2, 3^{e_1}, 4^{e_1}\}$ *and* $\{3^{e_2}, 4^{\neg e_1}\}$ *is* $\{2, 3^{e_1 \vee e_2}, 4\}$.

Definition 2.4.2 (V-set intersection) *The* intersection *of two v-sets is a v-set of their shared elements annotated with the conjunction of their presence conditions, i.e.,* $X_1 \cap X_2 = \{x^{e_1 \wedge e_2} \mid x^{e_1} \in X_1, x^{e_2} \in X_2, sat(e_1 \wedge e_2)\}$.

Definition 2.4.3 (V-set cross product) *The* cross product *of two v-sets is a pair of every two elements of them annotated with the conjunction of their presence conditions.* $X_1 \times X_2 = \{(x_1, x_2)^{e_1 \wedge e_2} \mid x_1^{e_1} \in X_1, x_2^{e_2} \in X_2\}$

Definition 2.4.4 (V-set equivalence) *Two v-sets are* equivalent*, denoted by* $X_1 \equiv X_2$*, iff* $\forall x^e \in (\downarrow (X_1) \cup \downarrow (X_2)), x^{e_1} \in\downarrow (X_1), x^{e_2} \in\downarrow (X_2) . e_1 \equiv e_2, e \equiv e_1$*, i.e., they both cover the same set of elements and the presence conditions of elements from the two normalized v-sets are equivalent.*

2.4.1 Variational Set Configuration

[vset configuration.]

2.5 The Formula Choice Calculus

[formula choice calculus]

The choice calculus [29, 14] is a metalanguage for describing variation in programs and its elements such as data structures [30, 15]. In the choice calculus, variation is

represented in-place as choices between alternative subexpressions. For example, 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 may be set to boolean values where `true` indicates the left alternative and `false` indicates the right alternative, we have: (1) $1 + 3 + 5$, when A and B are `true`, (2) $1 + 4 + 5$, when A is `true` and B is `false`, (3) $2 + 3 + 6$, when A is `false` and B is `true`, and (4) $2 + 4 + 6$, when A and B are `false`.

The formula choice calculus [20] 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, when $A \vee B$ evaluates to `true` and (2) 1, when $A \vee B$ evaluates to `false`.

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

Chapter 3: The Variational Database Framework

[needs. must have configuration.]

3.1 Variational Schema

[vsch]

3.1.1 Variational Schema Configuration

[vsch configuration.]

3.2 Variational Table

[vtab]

3.2.1 Variational Table Configuration

[vtab configuration]

3.3 Variational Database

[vdb]

3.3.1 Variational Database Configuration

[vdb configuration]

3.4 Properties of a Variational Database Framework

[well-formed vdb properties.context-specific properties.]

[show that they hold for vdb.]

Chapter 4: The Variational Query Language

[vql]

4.1 Variational Relational Algebra

[vra]

4.1.1 VRA Configuration

[vra configuration]

4.1.2 VRA Semantics

[vra semantics]

4.1.3 VRA Type System

[type sys]

4.1.4 VRA Variation-Minimization Rules

[rules]

4.2 Variational Query Language Properties

[prop. show for vra.]

Chapter 5: Variational Database Use Cases

[add this chapter text.] [revise sections] [add some commented out text from vamos]

5.0.1 Variation in Space: Email SPL Use Case

Our first use case focuses on variation in space. It shows the use of VDB to encode the variational information needs of a database-backed SPL. We consider an email SPL that has been used in several previous SPL research projects (e.g. [4, 3]). Our use case is formed by systematically combining two pre-existing works: (1) We use Hall's decomposition of an email system into its component features [17] as high-level specification of a SPL. (2) We use the Enron email dataset[1] as a realistic email database. In combining these works, we show how variation in space in an email SPL requires corresponding variation in a supporting database, how we can link the variation in the software to variation in the database, and how all of these variants can be encoded in a single VDB.

5.0.1.1 Variation Scenario: An Email SPL

The email SPL consists of the following features from [17]: [fix citet] *addressbook*, users can maintain lists of known email addresses with corresponding aliases, which may be used in place of recipient addresses; *signature*, messages may be digitally signed and verified using cryptographic keys; *encryption*, messages may be encrypted before sending and decrypted upon receipt using cryptographic keys; *autoresponder*, users can enable automatically generated email responses to incoming messages; *forwardmessages*, users can forward all incoming messages automatically to another address; *remailmessage*, users may send messages anonymously; *filtermessages*, incoming messages can be filtered according to a provided white list of known sender address suffixes; and *mailhost*, a list of known users is maintained and known users may retrieve messages on demand while messages sent to unknown users are rejected.

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

Table 5.1: Original Enron email dataset schema.

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

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

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

5.0.1.2 Generating V-Schema of the Email SPL VDB

To produce a v-schema for the email VDB, we start from plain schema of the Enron email dataset shown in Table 5.1, then systematically adjust its schema to align with the information needs of the email SPL described by [17]. [fix citet] The *employeelist* table contains information about the employees of the company. The *messages* table contains information about the email messages. The *recipientinfo* table contains information about the recipient of a message. The *referenceinfo* table contains messages that have been referenced in other email messages. This table simply backs up the emails.

[fix and include table]

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

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

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

A main focus of Hall's decomposition [17] is on the many feature interactions. Several of the features may interact in undesirable ways if special precautions are not taken. For example, any combination of the *forward_msg*, *remail_msg*, and *autoresponder* features can trigger an infinite messaging loop if users configure the features in the wrong way; preventing this creates an information need to identify auto-generated emails, which is realized in the variational schema by attributes like *is_forward_msg* and *is_autoresponse*.

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

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

5.0.2 Email Query Set

To produce a set of queries for the email SPL use case, we collected all of the information needs that we could identify in the description of the email SPL by [17]. [fix citet]. In order to make the information needs more concrete, we viewed the requirements of the email SPL mostly through the lens of constructing an email header. An email header includes all of the relevant information needed to send an email and is used by email

systems and clients to ensure that an email is sent to the right place and interpreted correctly. Although there is obviously other infrastructure involved, the fundamental information needs of an email system can be understood by considering how to construct email headers.

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

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

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

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

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

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

We can construct a query that retrieves the required header information whether *filtermessages* is enabled or not by combining Q_{basic} and Q_{filter} in a choice, as $Q_{bf} = filtermessages\langle Q_{filter}, Q_{basic}\rangle$. Although we do not show the process in this paper, we can use equivalence laws from the choice calculus [14, 20] to factor commonalities out of choices and reduce redundancy in queries like Q_{bf}. The other single-feature queries are written similarly.

Besides single-feature queries, we also provide queries that gather information needed to identify and address the undesirable feature interactions described by [17]. [fix citet] Out of Hall's 27 feature interactions, we determined 16 of them to have corresponding information needs related to the database; 2 of the interactions require 2 separate queries to resolve. Therefore, we define and provide 18 queries addressing all 16 of the relevant feature interactions. As before, we deduced the information needs through the lens of constructing an email header; in these cases, the header would correspond to an email produced after successfully resolving the interaction. However, some interactions can only be detected but not automatically resolved. In these cases, we constructed a query that would retrieve the relevant information to detect and report the issue.

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

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

the single-feature Q_{sig} query, but it is similar to other single-feature queries shown above.

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

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

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

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

sage or not. Therefore, we can resolve this interaction by extending the single-feature query for *filtermessages* to Q'_{filter}.

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

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

5.0.3 Variation in Time: Employee Use Case

Our second use case focuses on variation in time by demonstrating the use of a VDB to encode an employee database evolution scenario systematically adapted from [22] [fix citet] and populated by a dataset that is widely used in databases research.[2]

5.0.3.1 Variation Scenario: An Evolving Employee Database

[fix and include table]

[22] [fix citet] describe an evolution scenario in which the schema of a company's employee management system changes over time, yielding the five versions of the schema shown in Table **??**. In V_1, employees are split into two separate relations for engineer and non-engineer personnel. In V_2, these two tables are merged into one relation, *empacct*. In V_3, departments are factored out of the *empacct* relation and into a new *dept* relation to reduce redundancy in the database. In V_4, the company decides to start collecting more personal information about their employees and stores all personal information in the new relation *empbio*. Finally, in V_5, the company decides to decouple salaries from job titles and instead base salaries on individual employee's qualifications and performance; this leads to dropping the *job* relation and adding a new *salary* attribute to the *empacct* relation. This version also separates the *name* attribute in *empbio* into *firstname* and *lastname* attributes.

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

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

This yields the feature model:

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

5.0.3.2 Generating V-Schema of the Employee VDB

[fix and include table]

The v-schema for this scenario is given in Table **??**. It encodes all five of the schema versions in Table **??** and was systematically generated by the following process. First, generate a universal schema from all of the plain schema versions; the universal schema contains every relation and attribute appearing in any of the five versions. Then, annotate the attributes and relations in the universal schema according to the versions they are present in. For example, the *empacct* relation is present in versions V_2–V_5, so it will be annotated by the feature expression $V_2 \vee V_3 \vee V_4 \vee V_5$, while the *salary* attribute within the *empacct* relation is present only in version V_5, so it will be annotated by simply V_5. Since the presence conditions of attributes are implicitly conjuncted with the presence condition of their relation, we can avoid redundant annotations when an attribute is present in all instances of its parent relation. For example, the *empbio* relation is present in $V_4 \vee V_5$, and the *birthdate* attribute is present in the same versions, so we do not need to redundantly annotate it.

5.0.4 Employee Query Set

For this use case, we have a set of existing plain queries to start from. [22] [fix citet] provides 12 queries to evaluate the Prima schema evolution system. We adapt these queries to fit our encoding of the employee VDB described in Section 5.0.3. 9 of these queries have one variant, 2 have two variants, and 1 has three variants.

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

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

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

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

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

As another example, the following query realizes the intent to "return the name of the manager of department d001" during the time frame of V_3–V_5: $Q_3 = V_3 \vee V_4 \vee V_5\langle\pi_{name,firstname,lastname}\,(V_3\langle empacct, empbio\rangle \bowtie_{empno=managerno}\,(\sigma_{deptno=\text{“}d001\text{”}}\,dept)), \varepsilon\rangle$. Note that even though the attributes *name*, *firstname*, and *lastname* are not present in all three of the variants corresponding to V_3–V_5, the VRA encoding permits omitting presence conditions that can be completely determined by the presence conditions of the corresponding relations or attributes in the variational schema. So, Q_3 is equivalent to the following query in which the presence conditions of the attributes from the variational schema are listed explicitly in the projection: $Q_3' = V_3 \vee V_4 \vee V_5\langle\pi_{name^{V_3\vee V_4},firstname^{V_5},lastname^{V_5}}\,(V_3\langle empacct, empbio\rangle \bowtie_{empno=managerno}\,(\sigma_{deptno=\text{“}d001\text{”}}\,de$

Allowing developers to encode variation in v-queries based on their preference makes VRA more flexible and easy to use. Also, v-queries are statically type-checked to ensure that the variation encoded in them does not conflict the variation encoded in the v-schema.

Chapter 6: Variational Database Management System (VDBMS)

[vdbms]

6.1 Implemented Approaches

[apps]

6.2 Experiments

[exp.]

Chapter 7: Related Work

[related work! have to work on this!]

7.1 Instances of Variation in Databases

[schema evolution. database versioning. data integration. data provenance.]

7.2 Instances of Database Variation Resulted from Software Development

[SPL. data model. query.]

7.3 Variational Research

[blah]

Chapter 8: Conclusion

[conclusion]

Bibliography

[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.

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

[4] 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.

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

[6] Parisa Ataei, Qiaoran Li, and Eric Walkingshaw. Should variation be encoded explicitly in databases? In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS'21, New York, NY, USA, 2021. Association for Computing Machinery.

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

[8] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[9] 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.

[10] Cristina De Castro, Fabio Grandi, and Maria Rita Scalas. Schema versioning for multitemporal relational databases. *Information Systems*, 22(5):249 – 290, 1997.

[11] Badrish Chandramouli, Johannes Gehrke, Jonathan Goldstein, Donald Kossmann, Justin J. Levandoski, Renato Marroquin, and Wenlei Xie. READY: completeness is in the eye of the beholder. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[12] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, 2001.

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

[14] 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.

[15] 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.

[16] Mina Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. Clams: Bringing quality to data lakes. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2089–2092, New York, NY, USA, 2016. Association for Computing Machinery.

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

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

[19] 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.

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

[21] Edwin McKenzie and Richard Thomas Snodgrass. Schema evolution and the relational algebra. *Inf. Syst.*, 15(2):207–232, May 1990.

[22] 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.

[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.

[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 *13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 297–306. Gesellschaft für Informatik (GI), 2009.

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

[26] Micheal Stonebraker, Dong Deng, and Micheal L. Brodie. Database decay and how to avoid it. In *Big Data (Big Data), 2016 IEEE International Conference*. IEEE, 2016.

[27] U. Störl, D. Müller, A. Tekleab, S. Tolale, J. Stenzel, M. Klettke, and S. Scherzinger. Curating variational data in application development. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1605–1608, 2018.

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

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

[30] 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.

APPENDICES