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)

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

Chapter 1: Introduction

Managing variation in databases is a perennial problem in database literature and appears in different forms and contexts [30, 8, 12, 17, 9] and it is unavoidable [29]. 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 [22, 11, 6, 28, 23], data integration [14], and database versioning [10, 20]. However, there are no generic solution that addresses all 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 variation, and *variational relational algebra*, a query language for our framework that allows for information extraction from a variational database. The framework is generic because it can encode any kind of variation in databases. Additionally and more importantly, it is designed such that it can satisfy 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 [30, 8, 12, 17, 9]. 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 [31]. 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 [22, 11, 6, 28, 23], data integration [14], and database versioning [10, 20]. 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 [22, 11, 6, 28, 23]. 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 [22, 11, 6, 28] or keeping an external file of time-line history of changes applied to the database [23].

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) [13], a different database schema is required for each client due to client's different business requirements and environments [27]. 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 [27, 25, 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 [19]. 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 [29]; impacts DBAs, data analysts, and developers significantly [19]; 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 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. refs here have changed.

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

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*)

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.

refs here have changed. mot to mot-basic

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 [19].

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 [19].

The situation is further complicated since the basic and educational schemas

are interdependent: 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. And additionally, developed approaches of converting non-variational components into variational counterparts by using the introduced encoding of variation.

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 techniques 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

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

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

Figure 2.1: Syntax of relational algebra.

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. 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 and Encoding

[have to revise] [define oplus in fig as syntactic sugar.] [ref mot to ref mot-ex]

We encode variability in terms of *features*. The *feature space*, $\mathbf{F}$, of a variational database is a closed set of boolean variables called features. A feature $f \in \mathbf{F}$ can be enabled (i.e., $f =$ `true`) or disabled ($f =$ `false`). Features describe the variability 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 may represent different data sources.

Features are used at variation points to indicate which variants a particular element belongs to. Thus, enabling or disabling each of the features in the feature set 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 (and the rest are disabled). This database variant contains relation schemas in the yellow cells of Table 1.1. We refer to a variant by the configuration that produces it, e.g., variant

$\{V_2, T_3, edu\}$ refers to the variant produced by applying that configuration.

When describing variation points in the database, we need to refer to subsets of the configuration space. We do this with 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 where the *edu* feature is disabled, 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 the boolean value given by configuration c and then simplifies the propositional formula to a boolean value. For example, $\mathbb{E}[\![f_1 \vee f_2]\!]_{\{f_1\}} = \texttt{true} \vee \texttt{false} = \texttt{true}$, while $\mathbb{E}[\![f_1 \vee f_2]\!]_{\{\}} = \texttt{false} \vee \texttt{false} = \texttt{false}$. Additionally, in Figure 2.2, we define a binary *equivalence relation* ($\equiv$) relation on feature expressions corresponding to logical equivalence, and unary *sat* and *unsat* predicates that determine whether a feature expression is satisfiable or unsatisfiable, respectively.

To incorporate feature expressions into the database, we *annotate* 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 its *presence condition* since it determines the condition (set of configurations) under which the element is present in the database. For example, assuming $\mathbf{E} = \{f_1, f_2\}$, the annotated number $2^{f_1 \vee f_2}$ is present in variants $\{f_1\}$, $\{f_2\}$, and $\{f_1, f_2\}$ but not in variant $\{\}$. The operation

Feature expression syntax:

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

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.

$pc(x^e) = e$ returns the presence condition of an annotated element.

No matter the context, features often have a relationship with each other that constrains the set of possible 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*. That is, a configuration c is only valid if $\mathbb{E}[\![m]\!]_c = \texttt{true}$. For example, the restriction that at a given time only one of temporal features V_1–V_5 can be enabled is represented by the feature model $V_1 \oplus V_2 \oplus V_3 \oplus V_4 \oplus V_5$, where $f_1 \oplus f_2 \oplus \ldots \oplus f_n$ is syntactic sugar for $(f_1 \wedge \neg f_2 \wedge \ldots \wedge \neg f_n) \vee (\neg f_1 \wedge f_2 \wedge \ldots \wedge \neg f_n) \vee (\neg f_1 \wedge \neg f_2 \wedge \ldots \wedge f_n)$, that is, the features are mutually exclusive.

2.4 Variational Set

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.

[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 [16, 33, 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 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 [32, 15] is a metalanguage for describing variation in programs and its elements such as data structures [33, 16]. 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 [21] 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.]

the following is taken from VaMoS

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

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

1. There is at least one valid configuration of the feature model m: $sat(m)$
2. Every relation r is present in at least one configuration of the variational schema: $\forall r \in S, sat(m \wedge pc(r))$
3. Every attribute a in every relation r is present in at least one configuration of the variational schema: $\forall a \in r, \forall r \in S, sat(m \wedge pc(r) \wedge pc(a))$

4. If S_c denotes the expected plain relational schema for configuration c of the variational schema S, then configuring the variational schema with that configuration, written $[\![S]\!]_c$, actually yields that variant: $\forall c \in \mathbf{C}, [\![S]\!]_c = S_c$

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

1. Every tuple u in relation r is present in at least one variant: $\forall u \in r, \forall r \in S, sat(m \wedge pc(r) \wedge pc(u))$

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

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

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 props hold]

In this chapter, we describe how we systematically generated two variational databases from real world scenarios where variation appears in databases. We take a scenario where variation over either time or space exists in the database, use the schema variants to generate the variational schema, and attach feature expressions to tables and tuples to populate the VDB with data for each use case.

Additionally, variation in software affects not only databases but also how developers and database administrators interact with databases. Since different software variants have different information needs, developers must often write and maintain different queries for different software variants. Moreover, even if a particular information need is similar across variants, different variants of a query may need to be created and maintained to account for structural differences in the schema for each variant. Creating and maintaining different queries for each variant is tedious and error-prone, and potentially even intractable for large and open-ended configuration spaces, such as most open-source projects [29].

Thus, for each use case we present a set of variational queries and we illustrate how VRA realizes the information needs of the different variants of the database and potentially the corresponding software systems. It achieves this level of expressiveness by accounting for variation explicitly and linking variation in software and

databases to queries by using the same feature names and configuration space. We present only a sample of the queries here, yet we provide the full query sets in the link provided in Section **??**. online and will be linked to throughout this section. [fix link] The full query sets capture all of the information needs described in the papers that we base our variation scenarios on. It is important to note that this makes our query sets potentially biased toward queries containing more variation points since the focus of the papers is on variational parts of the system. A complete query set, capturing *all* information needs for each scenario might contain more plain queries, that is, queries that perform the same way over all variants. However, we do not believe this bias is harmful for the role the case studies are intended to serve, namely, motivating and evaluating variational database systems. For this role, queries that contain variation are more useful than plain queries, and additional plain queries can likely be more easily generated if needed.

[fix link] We distribute the v-queries in two formats: (1) VRA, encoded in the format used by our VDBMS tool, and (2) plain SQL queries with embedded `#ifdef` annotations to capture variation points.[1] The SQL format provides queries for studying variational data independently of VDBMS tool and will be more immediately useful for other researchers studying variational data independently of our VDBMS tool, but we use VRA in this paper for its brevity because it is much more concise.

[add details] The use case introduced in Section 5.1 illustrates variation in

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

databases over space. It illustrates generating a VDB from a SPL that produces email systems. It also demonstrates how variational queries can be used to address the inquiries of the email systems from the database. On the other hand, the use case introduced in Section 5.2 illustrates variation in databases over time. It illustrates generating a VDB from the schema evolution of an employee database and popular queries used to extract employees information from the VDB.

the following is from VaMoS intro regarding link.

We distribute the VDBs, SQL scripts for generating them, and queries of our use cases.[2] We distribute the VDBs in both MySQL and Postgres in two forms, one intended for use with our VDBMS tool, and one intended for more general-purpose research on variation in databases. We distribute the variational queries as simple `#ifdef`-annotated SQL files to promote their broad reuse in the design and evaluation of other systems for managing variational relational data.

5.1 Variation in Space: Email SPL Use Case

In our first case study, we focus on variation that occurs in "space", that is, where multiple software variants are developed and maintained in parallel. In software, variation in space corresponds to a SPL, where many distinct variants (products) can be produced from a single shared code base by enabling or disabling features. A variety of representations and tools have been developed for indicating which code belongs to which feature(s) and supporting the process of configuring a SPL

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

to obtain a particular variant.

Naturally, different variants of a SPL have different information needs. For example, an optional feature in the SPL may require a corresponding attribute or relation in the database that is not needed by the other features in the SPL. Currently, there is no good solution to managing the varying information needs of different variants at the level of the database. One possible solution is to manually maintain a separate database schema for each variant of the SPL. This works for some SPLs where the number of products is relatively small and the developer has control over the configuration process. However, it does not scale to open-source SPLs or other scenarios where the number of products is large and/or configuration is out of the developer's hands. Another possible solution is to use and maintain a single universal schema that includes all of the relations and attributes used by any feature in the SPL. In this solution, every product will use the same database schema regardless of the features that are enabled. This solves the problem of scaling to large numbers of products but is dangerous because it means that potentially several attributes and relations will be unused in any given product. Unused attributes will typically be populated by `NULL` values, which are a well-known source of errors in relational databases [1].

VDBs solve the problem by allowing the structure of a relational database to vary in a synchronous way with the SPL. Attributes and relations may be annotated by presence conditions to indicate in which feature(s) those attributes and relations are needed. An implementation of the VDB model might use a universal schema under-the-hood to realize VDBs on top of a standard relational

database management system (indeed, this is exactly how our prototype VDBMS implementation works), but by capturing the variation in the schema explicitly, we can validate (potentially variational) queries against the relevant variants of the variational schema to statically ensure that no `NULL` values will be referenced.

The email SPL use case 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. [5, 3]). It develops a variational schema that captures the information needs of a SPL based on Hall's decomposition of an email system into its component features [18]. The email SPL has been used in several previous SPL research projects (e.g. [4, 3]). The variational email database is populated using the Enron email dataset, adapted to fit our variational schema [26]. 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 [18] as high-level specification of a SPL.

2. We use the Enron email dataset[3] as a source of 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.

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

5.1.1 Variation Scenario: An Email SPL

The email SPL consists of the following features from Hall [18]:

- *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.

Note that Hall's decomposition separates *signature* and *encryption* into two features each (corresponding to signing and verifying, encrypting and decrypting).

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*)

Since these pairs of features must always be enabled together and they are so closely conceptually related, we reduce them to one feature each for simplicity.

The listed features are used in presence conditions within the v-schema for the email VDB, linking the software variation to variation in the database. In the email SPL, each feature is optional and independent, resulting in the simple feature model $m_{en} = \texttt{true}$, given as a feature expression. The feature model m_{en} is used as the root presence condition of the variational schema for the email VDB, implicitly applying it to all relations, attributes, and tuples in the database.

5.1.2 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 Hall [18]. The *employeelist* table contains information about the employees of the company including the employee identification number (*eid*), their first name and last name (*firstname* and *lastname*), their primary email address (*email_id*), alternative email addresses (e.g. *email2*), a path to the folder that contains their data (*folder*), and their last status in the company (*status*). The *messages* table contains infor-

Table 5.2: V-schema of the email VDB with feature model ? Presence conditions are colored blue for clarity.

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

mation about the email messages including the message ID (*mid*), the sender of the message (*sender*), the date (*date*), the internal message ID (*message_id*), the subject and body of the message (*subject* and *body*), and the exact folder of the email (*folder*). The *recipientinfo* table contains information about the recipient of a message including the recipient ID (*rid*), the message ID (*mid*), the type of the message (*rtype*), and the email address of the recipient (*rvalue*). The *referenceinfo* table contains messages that have been referenced in other email messages , for example, in a forwarded message; it contains a reference-info ID (*rid*), the message ID (*mid*), and the entire message (*reference*). This table simply backs up the emails.

[fix caption.]

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 5.2.

For example, consider the *signature* feature. In the software, implementing this feature requires new operations for signing an email before sending it out and for verifying the signature of a received email. These new operations suggest new information 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. For example, 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 [18] 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*. As another example consider the interaction that occurs between the *signature* and *remail_msg* features: the *remail_msg* feature enables anonymously sending messages by replacing the sender with a pseudonym, but this prevents the recipient from being able to verify a signed email. Furthermore, consider the interaction that occurs between the *signature* and *forward_msg* features: if Sarah signs a message and sends it to Ina, and Ina forwards the message to Philippe, then the signature verification operation may incorrectly interpret Ina as the sender rather than Sarah and fail to verify the message.

For each feature, we (1) enumerated the operations that must be supported both to implement the feature itself and to resolve undesirable feature interactions, (2) identified the information needs to implement these operations, and (3) extended the variational schema to satisfy these information needs. We make similar changes made to accommodate all features and their interactions.

For brevity, we omit some attributes and relations from the original schema that are irrelevant to the email SPL described by Hall [18], such as the *referenceinfo* relation and alternative email addresses.

[fix distribution] We distribute the variational schema for the email VDB in two formats. First, we provide the schema in the encoding used by our prototype VDBMS tool.[4] Second, we provide the v-schema 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

[4] usecases/space-emailSPL/schema

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.

5.1.3 Populating the Email SPL VDB

The final step to create the email VDB is to populate the database with data from the Enron email dataset, adapted to fit our v-schema [26]. For evaluation purposes, we want the data from the dataset to be distributed across multiple variants of the VDB. To simulate this, we identified five plausible configurations of the email SPL, which we divide the data among. The five configurations of the email SPL we considered are:

- *basic email*, which includes only basic email functionality and does not include any of the optional features from the SPL.
- *enhanced email*, which extends *basic email* by enabling two of the most commonly used email features, *forwardmessages* and *filtermessages*.
- *privacy-focused email*, which extends *basic email* with features that focus on privacy, specifically, the *signature*, *encryption*, and *remailmessage* features.
- *business email*, which extends *basic email* with features tailored to an environment where most emails are expected to be among users within the same business network, specifically, *addressbook*, *signature*, *encryption*, *autoresponder*, and *mailhost*.

- *premium email*, in which all of the optional features in the SPL are enabled.

For all variants, any features that are not enabled are disabled.

The original Enron dataset has 150 employees with 252,759 email messages. We load this data into the *employeelist* and *messages* tables defined in Section 5.1.2, initializing all attributes that are not present in the original dataset to `NULL`.

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

For the *messages* table, we again create five views corresponding to each of the variants. Each tuple is added to the view of the variant that contains the message's sender, which updates the tuple's presence condition accordingly. The *messages* table also contains several additional attributes corresponding to optional features, which we populate in a systematic way. We set *is_signed* to `true` if the

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

5.1.4 Email Query Set

To produce a set of queries for the email SPL use case, we collected all of the information needs that we could identify in the description of the email SPL by Hall [18].. In order to make the information needs more concrete, we viewed the requirements of the email SPL mostly through the lens of constructing an email header. An email header includes all of the relevant information needed to send an

email and is used by email systems and clients to ensure that an email is sent to the right place and interpreted correctly. More specifically, the email header includes the sender and receiver of the email, whether an email is signed and the location of a signature verification key, whether an email is encrypted and the location of the corresponding public key, the subject and body of the email, the mail host it belongs to, whether the email should be filtered, and so on. Although there is obviously other infrastructure involved, the fundamental information needs of an email system can be understood by considering how to construct email headers that ensures the email would get where it needs to go and be interpreted correctly on the other end.

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

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

We start by presenting the query to assemble the basic email header, Q_{basic}.

This corresponds to the information need of a system with no features enabled. We use X to stand for the specific message ID (*mid*) of the email whose header we want to construct.

$$Q_{basic} = \pi_{sender,rvalue,subject,body}\, mes\_rec$$

$$mes\_rec \leftarrow (\sigma_{mid=X}\, messages) \bowtie recipientinfo$$

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

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

$$Q_{filter} = \pi_{sender,rvalue,suffix,subject,body}\, temp$$

$$temp \leftarrow mes\_rec\_emp \bowtie filter\_msg$$

$$mes\_rec\_emp \leftarrow mes\_rec \bowtie_{rvalue=email\_id}\ employeelist$$

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 $Q_{bf} = \mathit{filtermessages}\langle Q_{filter}, Q_{basic}\rangle$. Although we do not show the process in this paper, we can use equivalence laws from the choice calculus [15, 21] to factor commonalities out of choices and reduce redundancy in queries like Q_{bf}. The other single-feature queries are written similarly.

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

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

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

Besides single-feature queries, we also provide queries that gather information needed to identify and address the undesirable feature interactions described by Hall [18]. Out of Hall's 27 feature interactions, we determined 16 of them to have corresponding information needs related to the database; 2 of the interactions require 2 separate queries to resolve. Therefore, we define and provide 18 queries addressing all 16 of the relevant feature interactions. As before, we deduced the

information needs through the lens of constructing an email header; in these cases, the header would correspond to an email produced after successfully resolving the interaction. However, some interactions can only be detected but not automatically resolved. In these cases, we constructed a query that would retrieve the relevant information to detect and report the issue.

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

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

other single-feature queries shown above.

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

The query Q_{sf} also resolves another consequence of the interaction between these two features. This time Sam successfully verifies message X and forwards it to Sam2 which changes the header in the system such that it states message X has been successfully verified, thus, the message could be altered by hackers while it is being forwarded (Hall's interaction #27). The system can use Q_{sf} to generate the correct header in this scenario again.

Some feature interactions require more than one query to satisfy their information need. 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.

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

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

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

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.

5.2 Variation in Time: Employee Use Case

In our second case study, we focus on variation that occurs in "time", that is, where the software variants are produced sequentially by incrementally extending and modifying the previous variant in order to accommodate new features or changing business requirements. Although new variants conceptually replace older variants, in practice, older variants must often be maintained in parallel; external dependencies, requirements, and other issues may prevent clients from updating to the latest version. Variation in software over time directly affects the databases such software depends on [29], and dealing with such changes is a well-studied problem in the database community known as *database evolution* [24].

Although research on database evolution has produced a variety of solutions for managing database variation over time, these solutions do not treat variation as an orthogonal property and so cannot also accommodate variation in space. The goal of our work on variational databases is not to directly compete with database evolution solutions for time-only variation scenarios, but rather to present a more general model of database variation that can accommodate variation in both time and space, and that integrates with related software via feature annotations.

We demonstrate variation in time by using a VDB to encode an employee database evolution scenario systematically adapted from Moon et al. [23] and pop-

Table 5.3: Evolution of an employee database schema.

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

ulated by a dataset that is widely used in databases research.[5]

5.2.1 Variation Scenario: An Evolving Employee Database

[fix caption]

Moon et al. [23] 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 5.3. In V_1, employees are split into two separate relations for engineer and non-engineer personnel. In V_2, these two tables are merged into one relation, *empacct*. In V_3, departments are factored out of the *empacct* relation and into a new *dept* relation to reduce redundancy in the database. In

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

V_4, the company decides to start collecting more personal information about their employees and stores all personal information in the new relation *empbio*. Finally, in V_5, the company decides to decouple salaries from job titles and instead base salaries on individual employee's qualifications and performance; this leads to dropping the *job* relation and adding a new *salary* attribute to the *empacct* relation. This version also separates the *name* attribute in *empbio* into *firstname* and *lastname* attributes.

We associate a feature with each version of the schema, named $V_1 \ldots V_5$. These features are mutually exclusive since only one version of the schema is valid at a time. This yields the feature model m_{emp}. Also, note that the feature model represent a restriction on the entire database.

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

5.2.2 Generating V-Schema of the Employee VDB

[fix caption]

The v-schema for this scenario is given in Table 5.4. It encodes all five of the schema versions in Table 5.3 and was systematically generated by the following process. First, generate a universal schema from all of the plain schema versions; the universal schema contains every relation and attribute appearing in any of the

Table 5.4: Employee v-schema with feature model.

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

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. The overall variational schema will be annotated by the feature model m_{emp}, described in Section 5.2.1. Since the presence conditions of attributes are implicitly conjuncted with the presence condition of their relation that contains them, 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 *birthdate*.

[make sure it's consistent] Similar to the email SPL VDB, we distribute the variational schema for the employee VDB in two formats: First, we provide the schema in the encoding used by our prototype VDBMS tool.[6] Second, we provide a direct encoding in SQL that generates the universal schema for the VDB in

[6] usecases/time-employee/schema/EmployeeSchema.hs

either MySQL or Postgres.[7] The variability of the schema is embedded within the employee VDB[8] using the same encoding as described at the end of Section 5.1.2.

5.2.3 Populating the Employee VDB

Finally, we populate the employee VDB using data from the widely used employee database linked to in this subsection's lede. This database contains information for $240,124$ employees. To simulate the evolution of the database over time, we divide the employees into five roughly equal groups based on their hire date within the company. For example, the first group consists of employees hired before $1988-01-01$, while the second group contains employees hired from $1988-01-01$ to $1991-01-01$. Each group is assumed to have been hired during the lifetime of a particular version of the database, and is therefore added to that version of the database and *also* to all subsequent versions of the database. This simulates the fact that as a database evolves, older records are typically forward propagated to the new schema [24]. Thus, V_5 contains the records for all $240,124$ employees, while older versions will contain progressively fewer records. The final employee VDB has $954,762$ employee due to this forward propagation, despite having the same number of employees as the original database.

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

[7] usecases/time-employee/database/create
[8] usecases/time-employee/database/withSchema

v-schema.

[check] We provide SQL scripts of required queries to automatically generate the employee VDB. [9] We also provide the final employee VDB in four flavors: both with and without the embedded schema, and in both cases, encoded in MySQL and PostgreSQL format.[10]

5.2.4 Employee Query Set

[make query names consistent] [make queryies consistent. check theyre correct.] [fix format]

For this use case, we have a set of existing plain queries to start from. Moon et al. [23] 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 5.2. [check consistent] We provide the queries in both the VRA format usable by VDBMS and as `#ifdef` annotated SQL, as described in Section 5.1.4.[11] 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:

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

[9] usecases/time-employee/database/build
[10] usecases/time-employee/database
[11] usecases/time-employee/queries

Note that the presence condition of the only attribute *salary* determines the presence condition of the resulting table.

In general and for simplicity, the shared part of presence conditions of projected attributes is factored out and applied to the entire table. Assume the returned table as a result of query has the schema $({a_1}^{e \wedge e_1}, {a_2}^{e \wedge e_2})$. The shared restriction can be factored out and applied to the entire table, i.e., $({a_1}^{e_1}, {a_2}^{e_2})^e$.

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.

The feature expression $V_3 \vee V_4 \vee V_5$ determines the database variants to be inquired. Since the schema of *empacct* and *job* tables are the same in variants V_3 and V_4 they both have the same query. Note that one could move the condition $V_3 \vee V_4 \vee V_5$ to the projected attribute which results in $empQ'_2$, however, this query is wrong because the last alternative of the choice projects attribute *salary* from an empty relation which is incorrect. It is important to understand that the behavior

of an empty relation is exactly the same as its behavior in relational algebra and one should be careful of using it in operations such as projection, selection, and join.

$$
\begin{aligned}
emp Q_2' = \pi_{salary^{V_3 \vee V_4 \vee V_5}} \\
(V_3 \vee V_4 \langle (\sigma_{empno=10004}\, empacct) \bowtie_{empacct.title=job.title} job \\
, V_5 \langle \sigma_{empno=10004}\, empacct, \varepsilon \rangle \rangle)
\end{aligned}
$$

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:

$$
\begin{aligned}
Q_3 = V_3 \vee V_4 \vee V_5 \langle \pi_{name,firstname,lastname} \\
(V_3 \langle empacct, empbio \rangle \bowtie_{empno=managerno} (\sigma_{deptno=\text{“}d001\text{”}}\, dept)), \varepsilon \rangle
\end{aligned}
$$

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:

$$Q'_3 = V_3 \vee V_4 \vee V_5\langle \pi_{name^{V_3 \vee V_4}, firstname^{V_5}, lastname^{V_5}}$$
$$(V_3\langle empacct, empbio\rangle \bowtie_{empno=managerno} (\sigma_{deptno=\text{“}d001\text{”}}\, dept)), \varepsilon\rangle$$

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

Finally, we want to briefly illustrate what queries look like in the `#ifdef`-annotated SQL format that we distribute as a potentially more portable and easy-to-use format for other researchers. Below is the query Q_3 in this format.

```
#ifdef V3 || V4 || v5
  #ifdef V3 || V4
SELECT name
  #else
SELECT firstname, lastname
  #endif
FROM
  #ifdef V3
    empacct
  #else
    empbio
```

```
  #endif
JOIN (SELECT dept WHERE deptno="d001")
  ON empno=manageno

#ifdef v3
SELECT salary
FROM empacct JOIN job ON empacct.title=job.title
WHERE empno=10004
#endif
```

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]

the following is taken from VaMoS

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

Expressiveness of explicit variation. The use cases in Section **??** and Section **??** show that by treating variation as an orthogonal concern and embedding it directly in databases and queries (via presence conditions and choices), one can encode data variation scenarios in both time and space. In fact, VDBs and v-queries are *maximally expressive* in the sense that any set of plain relational databases can be encoded as a single VDB and any set of plain queries over the variants of a VDB can be encoded as a v-query.[1]

The expressiveness of our approach is its main advantage over other ways to manage database variation. When working with a form of variation that already has its own specialized solution (e.g. schema evolution, data integration), the ex-

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

pressiveness of explicit variation is probably not worth the additional complexity. The expressiveness of explicit variation is most useful when working with a form of variation that is not well supported (e.g. query-level variation in SPLs), or when combining multiple forms of variation in one database (e.g. during SPL evolution).

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

reducing redundancy across clients.

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

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

It is possible that a more restrictive and/or coarse-grained form of variation in v-queries would make them easier to understand at the cost of increased redundancy and (potentially) reduced expressiveness. This tradeoff is one we already made when considering how to encode variation in the *content* of a VDB. Specifi-

cally, we do not support cell-level variation in a VDB (e.g. choices within individual cells). This does not reduce the expressiveness of content variation in VDBs since cell-level variation can be simulated by row variation, but it does increase redundancy since all non-varied cells in the row must be duplicated. Similarly, variation in queries could be restricted to expression-level choices, with no choices or annotations in conditions or attribute lists. This would likely make understanding individual query variants easier at the cost of increasing redundancy among the alternatives of each choice.

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

Analyzability of explicit variation. The relationship of our work to alternative approaches can be viewed through the lens of annotative vs. compositional variation, familiar to the SPL community [**?**]. VDBs and v-queries rely on generic annotations embedded directly in schemas and queries, respectively, while approaches from the databases community often express variation through separate artifacts, such as views [**?**]. Annotative vs. compositional representations often exhibit the same tradeoff between expressiveness and complexity described above: annotative variation tends to be general and expressive, while compositional variation tends to be more restrictive but support modular reasoning [**?**]. Traditionally,

another advantage of compositional approaches is that they are more analyzable thanks to the ability to analyze components separately (i.e. *feature-based* analysis [**?**]), a benefit shared by database views. However, in the last decade there has been a significant amount of work in the SPL community to improve the analyzability of annotative variation by analyzing whole variational artifacts directly (i.e. *family-based* analysis [**?**]). Although not presented here, we build directly on this body of work, especially work on variational typing [**?** **?**], to enable efficiently checking v-queries against all variants of a VDB, among other properties. Thus, the increased complexity of explicit variation annotations does not prevent us from verifying its correctness.

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. ISBN 978-3-642-21759-3.

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

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

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

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

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

[8] Parisa Ataei, 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. ISBN 9781450388245. doi: 10.1145/3442391.3442395. URL https://doi.org/10.1145/3442391.3442395.

[9] 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. URL http://cidrdb.org/cidr2015/Papers/CIDR15_Paper18.pdf.

[10] 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.

[11] Cristina De Castro, Fabio Grandi, and Maria Rita Scalas. Schema versioning for multitemporal relational databases. *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.

[12] 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. URL http://cidrdb.org/cidr2017/papers/p18-chandramouli-cidr17.pdf.

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

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

[15] 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.

[16] 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.

[17] 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. ISBN 9781450335317. doi: 10.1145/2882903.2899391. URL https://doi.org/10.1145/2882903.2899391.

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

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

[20] 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.

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

[22] Edwin 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.

[23] 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.

[24] John F Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383 – 393, 1995. ISSN 0950-

5849. doi: https://doi.org/10.1016/0950-5849(95)91494-K. URL http://www.sciencedirect.com/science/article/pii/095058499591494K.

[25] 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. ISBN 978-3-642-31095-9.

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

[27] 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.

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

[29] 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. doi: 10.1109/BigData.2016.7840584.

[30] 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. doi: 10.1109/ICDE.2018.00187.

[31] 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.

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

[33] 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