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 denotational semantics of the variational relational algebra, a query language for variational databases. 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.

drop the var pres prop at semantics if you don't have it

??

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

To my loving parents,

Sedigheh Bazrafshan and Seyed Jalal Ataei.

I count the days to embrace you both in my arms again

(after 2,109 days and still counting).

Chapter 1 Introduction

remember to adjust ref to requirements. it's a new chapter now.

Managing variation in databases is a perennial problem in database literature and appears in different forms and contexts [54, 10, 15, 26, 12] and it is unavoidable [52]. 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 [41, 14, 7, 51, 43], data integration [22], and database versioning [13, 33]. 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.

vamos. incorporate these into mot.

Just as variation is ubiquitous in software, it is also ubiquitous in the relational databases that software systems rely on. Database researchers have long studied different kinds of variation in databases, such as through work on database evolution [14, 41, 43], database versioning [13, 33], and data integration [22]. However, work in the databases community does not identify *variation* as a general, orthogonal concern that arises in many different contexts. This is a problem since it means that tools and techniques developed for one kind of database variation cannot be easily reused for another. More concretely for software developers, and especially software product line (SPL) practitioners, the lack of a general representation of database variation means that many kinds of variation that arise in software do not map cleanly onto corresponding variation encodings in the databases they use.

In contrast, SPL researchers have invested significant effort in studying *variation* (or *variability*) as a general phenomenon and concern in software. Although many kinds of software variation are possible, most can be roughly organized into variation in *time* or *space* [57]. Variation in *time* refers to the evolution of a system

and is addressed by revision control systems and configuration management [21], while variation in *space* refers to the simultaneous development and maintenance of related systems with different feature sets and is the traditional focus of research on SPLs [6]. Multiple lines of work in the SPL community have sought to develop general purpose representations of variation in software, such as delta-oriented programming [47] and the choice calculus [23], among others. These can be used to unify both kinds of variation in software, enabling reuse of analyses across dimensions and enabling new kinds of analyses that consider variation in both time and space simultaneously [57].

Because the existing data variation models developed in the databases community do not align with all of the variation scenarios that arise in software, SPL researchers have identified the need for more general encodings of variation in data models and database schema. To this end, they have developed encodings of data models that allow for arbitrary variation by annotating different elements of the model with features from the SPL [50, 48, 2]. However, these solutions address only variation in the data model but do not extend to the level of the data or queries. The lack of variation support in queries leads to unsafe techniques such as encoding different variants of query through string munging, while the lack of variation support in data precludes testing with multiple variants of a database at once.

In previous work, we have developed *variational databases (VDBs)* and *variational queries* [9, 8] to encode general variation in the representation and use of relational databases. Our work extends ideas developed in the SPL community

use the following paragraph and items for contribution maybe.

to relational databases. Conceptually, a VDB represents potentially many different plain relational databases at the same time. Similarly, a variational query represents potentially many different queries, each one corresponding to a variant of the VDB. Together, VDBs and variational queries enable safely and efficiently working with many variants of a relational database at once, and reliably integrating the variants of a database with the corresponding variants of an SPL. We are currently implementing these ideas in *VDBMS*, a practical implementation of variational databases as a lightweight wrapper on top of a traditional relational database management system.

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

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

end of from vamos. remember to read the commented out part too.

1.1 Motivation and Impact

from vldb. revise intro.

Managing variation in databases is a perennial problem that appears in different forms and contexts [54, 10, 15, 26, 12]. In databases, variation arises when several database instances, which may differ in schema, content, or constraints, conceptually represent the same database. Existing work on database variation focuses on specific kinds of variation such as schema evolution [41, 14, 7, 51, 43], data integration [22], and database versioning [13, 33]. These works provide solutions specific to the kinds of variation they address, but do not provide a general-purpose solution to managing variation in databases. This is a problem because new kinds of variation frequently arise and different kinds of variation often interact, and the existing solutions are often ill-suited to these scenarios.

Schema evolution is an example of a kind of variation in databases that is well-supported [41, 14, 7, 51, 43]. In the schema evolution scenario, a database's schema changes over time as the database is extended or refactored to support new information needs, and the goal is to automatically migrate data and queries to new versions of the database. Thus, schema evolution is a kind of *variation* over the dimension of time, and each version of the database can be viewed as a *variant* of the same database.

However, other kinds of database variation are less well supported. One example arises in the context of *software product lines* (SPLs) [**?**]. An SPL arises when the same code base is used to produce multiple different variants of a software system, customized with different sets of features or tailored for different clients. Naturally, the data requirements of each variant of an SPL may differ [50]. SPL researchers have developed various encodings that allow describing variation in the

data model among variants by annotating elements of the model with features from the SPL [50, 48, 2]. These solutions can generate a database schema variant for each software variant of the SPL. However, these solutions address only variation in the data model but do not extend to the level of 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.

Worse still is when multiple kinds of variation interact. Although structural variation over time is well-supported via schema evolution, and structural variation in "space" is partially supported by recent SPL research, there is no support for the inevitable evolution of an SPL's variational data model [31]. Nor do existing approaches support variation across all levels of a relational database: schema, queries, and content. In our previous work, we have argued that schema evolution, SPL-like variation, and other forms of database variation, such as data integration and database versioning, are all facets of a similar problem that can be addressed by *treating variation as a general and orthogonal concern* in relational databases [8, 9, 10]. A significant advantage of treating different kinds of variation uniformly is that it is easy to support the interaction of multiple kinds of variation, and to coordinate variation in structure with corresponding variation in queries and content. We illustrate these aspects throughout the paper using a motivating example described in Section 1.1.

In previous work, we have proposed the idea of *variational databases*, which incorporate variation as a general and orthogonal concern in relational databases [8,

9]. In this paper, we significantly expand on and realize this idea through the formalization and implementation of a *variational database management system* (VDBMS). Specifically, this paper makes the following contributions:

- We provide a formal model of *variational databases* (VDB), whose structure are given by a *variational schema* and whose content are given by *variational tables* (Section **??**).
- We define *variational relational algebra* as a query language for VDB, a *static type system* for ensuring that all variants of a query are compatible with the corresponding variants of the VDB, and a set of *minimization rules* for reducing the size of queries (Section **??**).
- We implement a prototype of VDBMS as a layer on top of a traditional DBMS (Section **??**) and evaluate this implementation on previously developed [10] use cases (Section **??**).

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 function-

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*)

alities 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 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 [31].

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 [31].

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.

1.2 Contributions and Outline of this Thesis

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.

write more details

Chapter 3 (*Variational Database Framework*) describes a variational database framework and an approach to encode variation both at the schema level (Section 3.1) and content level (Section 3.2), resulting in a variational database (VDB). Then, Section 3.3 defines properties of a well-formed VDB.

write this after you finish the chapter.

[Chapter 4]

Chapter 5 (*Variational Database Use Cases*) aims at guiding an expert through

generating a VDB and writing variational queries for a variation scenario where unfortunately the database variants do not exist and thus, generating the VDB requires the expert knowledge and cannot be automated. It details two such variation scenario and introduces two use cases of VDB, one over space (adopts the email SPL described by Hall [30] and explained in Section 5.1) and another over time (adopts the evolution of an employee schema described by Moon et al. [43] and explained in Section 5.2), in addition to how a VDB can store all database variants in a single database and how variational queries can capture various information needs over different database variants in a single query. It also describes how the VDBs were systematically generated and how the variational queries were adapted and adjusted from papers describing the variation scenario. The last section of this chapter, Section 5.3, discusses the question "should variation be encoded explicitly in databases?" through the lessons learned while generating these use cases.

write this after you finish the chapter.

[Chapter 6]

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

write this after you finish the chapter and do your research.

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

write this after you finish the chapter.

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 approaches used to add variation to them.

Throughout the thesis, we use types when defining concepts. A type is a set of possible values. For example, the type **Int** denotes all possible integers. In our formalization, we use the notation of $i \in \textbf{Int}$ to state that the variable i is of type **Int**. Types can be more general. Consider the type **Set a** that indicates the set all sets of values of type **a**. Here, **a** is type variable and stands for any possible type. Note that concrete types start with a capital letters but type variables do not. For example, the type **Set Int** is the type of sets of integers and it has values such as $\{1, 3, 4\}$ and $\{\ \}$. We also use type synonyms for simplicity. For example, instead of referring to **Set Int** we can give it a new name (**SetInt** = **Set Int**) and refer to it with the new name **SetInt**. Sometimes we must extend a type with an additional "bottom" element $\bot$ and to account for this extension at the type level we subscript the type with $\bot$. For example, $\mathcal{I}_\bot$ denotes the extension of the type $\mathcal{I}$ with $\bot$. Furthermore, sometimes we pair two types together. For

pair and function types

example, a variable of type **(Int, String)** can take the value $(3, "three")$. Finally, we use function types to provide the type signature of functions. For example, the function $id : \mathbf{a} \to \mathbf{a}$ states that the function id takes a value of type $\mathbf{a}$ and returns a value of type $\mathbf{a}$.

Throughout the thesis, we discuss relational concepts and their variational counterparts. For clarity or when it is unclear from context, we use an *underline* to distinguish a non-variational entity from its variational counterpart, both at the value level and the type level, e.g., $\underline{x}$ is a non-variational entity while x is its variational counterpart, if it exists.

Section 2.1 describes the database model of relational databases and the specification of the structure used to store the data [1]. Section 2.2 describes the relational algebra, a query language used to query relational databases [1]. Section 2.3 defines our encoding of the variation space used in variational database and how we describe parts of that space using propositional formulas of boolean variables [9, 8]. Finally, we introduce the 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 [25, 60, 8] and Section 2.5 describes the formula choice calculus used to incorporate variation into relational algebra [34].

2.1 Relational Databases

We introduce relational databases concepts, which variational databases are based on. Table 2.1 represents an example relational database instance corresponding to V_2 introduced in our motivating example in Table 1.1a. Intuitively, the data is represented in tables with rows of uniform structure where each row contains data about a specific object or sets of object [1]. Each table is associated with a relation and has a name, for example, the relation *job* contains rows specifying the salary of a position in a company. The columns form the structure of the table and are called attributes, for example, the relation *job* has two attributes: *title* and *salary*. The values of attributes are taken from a set of constants called the domain. For simplicity, we do not distinguish between different types of values such as strings, numbers, dates.[1] Finally, we differentiate between the database schema, which is the structure data is stored, and the database instance, which is the actual content of the database. For example, Table 2.1b and Table 2.1c illustrate the database instance while Table 2.1a shows the database schema. This differentiation can be viewed as the differentiation of types and values in programming languages. For example, x is variable of type *Int* and may have value 15.

We now shift our focus to the formal definitions of a relational 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

[1] This design decision was made to make proofs easier, however, our database system implementation distinguishes between different kinds of values and considers a unique domain for each.

Table 2.1: An example of a relational database corresponding to V_2 of our motivating example given in Table 1.1a.

(a) The schema of a relational database.

empacct (*empno*, *name*, *hiredate*, *title*, *deptname*)
job (*title*, *salary*)

(b) The *empacct* table.

empacct

empno	*name*	*hiredate*	*title*	*deptname*
10001	Georgi Facello	1986-06-26	Senior Engineer	Development
10002	Bezalel Simmel	1985-11-21	Staff	Sales
...	...	...	...	...
499998	Patricia Breugel	1993-10-13	Senior Staff	Finance
499999	Sachin Tsukuda	1997-11-30	Engineer	Production

(c) The *job* table.

job

title	*salary*
Assistant Engineer	61594
Senior Engineer	96646
...	...
Staff	77935
Technique Leader	58345

Relational database objects:

$$
\begin{array}{rcll}
a \in \textbf{AttrName} & & & \textit{Attribute Name} \\
r \in \textbf{RelName} & & & \textit{Relation Name} \\
\underline{u} \in \textbf{Tuple} & \coloneqq & (v_1, \ldots, v_l) & \textit{Tuple} \\
\underline{A} \in \textbf{Set AttrName} & \coloneqq & \{a_1, a_2, \ldots, a_l\} & \textit{Set of Attributes} \\
\underline{s} \in \underline{\textbf{RelSch}} & \coloneqq & r\,(\underline{A}) & \textit{Relation Schema} \\
\underline{U} \in \underline{\textbf{RelCont}} & \coloneqq & \{\underline{u}_1, \underline{u}_2, \ldots, \underline{u}_k\} & \textit{Relation Content} \\
\underline{t} \in \underline{\textbf{Table}} & \coloneqq & (\underline{s}, \underline{U}) & \textit{Table} \\
\underline{S} \in \underline{\textbf{Sch}} & \coloneqq & \{s_1, \ldots, s_n\}^e & \textit{Schema} \\
\underline{\mathcal{I}} \in \underline{\textbf{DBInst}} & \coloneqq & \{\underline{t}_1, \underline{t}_2, \ldots, \underline{t}_n\} & \textit{Database Instance}
\end{array}
$$

Relational database type synonyms:

$$
\begin{aligned}
\underline{\textbf{RelCont}} &= \textbf{Set Tuple} \\
\underline{\textbf{Table}} &= (\underline{\textbf{RelSch}}, \underline{\textbf{RelCont}}) \\
\underline{\textbf{Sch}} &= \textbf{Set } \underline{\textbf{RelSch}} \\
\underline{\textbf{DBInst}} &= \textbf{Set } (\underline{\textbf{RelSch}}, \underline{\textbf{RelCont}})
\end{aligned}
$$

Figure 2.1: Relational database objects and type synonyms.

relation schema is defined as $\underline{s} = r\,(a_1, \ldots, a_k)$ where each $a_i \in \mathbf{AttrName}$ is an *attribute* contained in the relation named r and **AttrName** is a fixed countably infinite set of attributes. We assume a total order $\leq_{\mathbf{AttrName}}$ on **AttrName**, and assume for simplicity that sets of attributes are sorted according to $\leq_{\mathbf{AttrName}}$.

Figure 2.1 defines the relational database objects and type synonyms compactly. The content of database $\underline{D}$ is stored in the form of *tuples*. A tuple $\underline{u}$ is a list of *values*. We do not distinguish between different types of values within a relational database. The order of values within a tuple correspond to the order of attributes in its corresponding relation schema, that is, given tuple $\underline{u} = (\underline{v}_1, \ldots, \underline{v}_k)$ in the relation with relation schema $r(a_1, \ldots, a_k)$, $\underline{v}_i$ corresponds to attribute $\underline{a}_i$. A *relation content*, $\underline{U}$, is the set of all tuples $\{\underline{u}_1, \ldots, \underline{u}_m\}$ corresponding to a particular relation. The operation $att(i)$ returns the attribute corresponding to index i in a tuple, implicitly looking up the attribute in the corresponding relation schema. A *table* $\underline{t} = (\underline{s}, \underline{U})$ is a pair of a relation schema and relation content. A *database instance*, $\underline{\mathcal{I}}$, of the database $\underline{D}$ with schema $\underline{S}$, is a set of tables $\{\underline{t}_1, \ldots, \underline{t}_n\}$ for each relation in $\underline{S}$. For brevity, when it is clear from context, we refer to a database instance as simply a *database*.

2.2 The SPC Relational Algebra

add type system

maybe add semantics later on

Having introduced relational databases we now shift gears into querying these databases, that is, extracting information from tables. For almost all relational query languages, the result of a query is a table called *result*. We base our vari-

ational query language on the SPC relational algebra. Three primitive operators form the SPC algebra: *selection*, *projection*, and *cross-product* (or Cartesian product) [1]. We introduce these operators through Example 2.2.1 by stating an intent and then building up a query to extract the information required by the intent.

Example 2.2.1. *Consider the database instance given in Table 2.1. We want to get a list of employees (by their names) whose salary is more than 65000 dollars. As the first step, we use the selection operator to get the tuples for all jobs with salaries that are more than 65000 dollars.*

$$\underline{q}_1 = \sigma_{salary \geq 65000}(job)$$

A sample of the results returned by the query $\underline{q}_1$ *is given in Table 2.2a. Next a set of tuples is created by taking the cross-product of* $\underline{q}_1$ *and empacct.*

$$\underline{q}_2 = \underline{q}_1 \times empacct$$

A sample of the results returned by the query $\underline{q}_2$ *is given in Table 2.2b. However, looking closely at these results there is no connection between an employee in the empacct relation and their salary in the job relation. Thus, we have to perform another selection to connect each employee with their title.*

$$\underline{q}_3 = \sigma_{empacct.title=job.title}(\underline{q}_2)$$

A sample of the results returned by the query $\underline{q}_3$ *is given in Table 2.2c. At this point, we are only interested in two attributes, that is, name and salary. Thus, we use projection to discard the unneeded columns.*

$$\underline{q}_4 = \pi_{name,salary}(\underline{q}_3)$$

A sample of the results returned by the query $\underline{q}_4$ *is given in Table 2.2d.*

The relational algebra that we use also includes standard set operations, a join

Table 2.2: Results of each step of building the final query in Example 2.2.1.

(a) Result of the query $\sigma_{salary \geq 65000}(job)$.

result

title	*salary*
Senior Staff	77935
Senior Engineer	96646
...	...
Staff	77935
Engineer	96646

(b) Result of the query $(\sigma_{salary \geq 65000}(job)) \times empacct$.

result

title	*salary*	*empno*	*name*	*hiredate*	*title*	*deptname*
Senior Engineer	96646	13094	Sanjay Servieres	1986-01-01	Engineer	Research
Staff	77935	16099	Mohan Ferretti	1987-09-20	Senior Staff	Human Resources
...	...	...	...	...	...	...
Engineer	80324	19162	Chinho Fadgyas	1986-05-19	Technique Leader	Production
Senior Staff	88070	22255	Kristian Merel	1986-09-12	Senior Engineer	Development

(c) Result of the query $\sigma_{empacct.title=job.title}((\sigma_{salary \geq 65000}(job)) \times empacct)$.

result

title	*salary*	*empno*	*name*	*hiredate*	*title*	*deptname*
Engineer	96646	13094	Sanjay Servieres	1986-01-01	Engineer	Research
Senior Staff	77935	16099	Mohan Ferretti	1987-09-20	Senior Staff	Human Resources
...	...	...	...	...	...	...
Senior Engineer	96646	22255	Kristian Merel	1986-09-12	Senior Engineer	Development
Staff	77935	43670	JoAnna Randi	1987-10-18	Staff	Marketing

(d) Result of the query $\pi_{name,salary}(\sigma_{empacct.title=job.title}((\sigma_{salary \geq 65000}(job)) \times empacct))$.

result

name	*salary*
Sanjay Servieres	96646
Mohan Ferretti	77935
...	...
Kristian Mere	96646
JoAnna Randi	77935

Operators:

$$\bullet \coloneqq < \mid \leq \mid = \mid \neq \mid > \mid \geq$$
$$\circ \coloneqq \cup \mid \cap$$

Variational conditions:

$$\underline{\theta} \in \underline{\textbf{Condition}} \coloneqq \texttt{true} \mid \texttt{false} \mid a \bullet k \mid a \bullet a \mid \neg\underline{\theta} \mid \underline{\theta} \vee \underline{\theta}$$

Relational queries:

$$\begin{array}{rcll} \underline{q} \in \underline{\mathbf{Q}} & \coloneqq & \underline{r} & \textit{Relation} \\ & \mid & \pi_{\underline{A}}\underline{q} & \textit{Projection} \\ & \mid & \sigma_{\underline{\theta}}\underline{q} & \textit{Selection} \\ & \mid & \underline{q} \times \underline{q} & \textit{Cross-Product} \\ & \mid & \underline{q} \circ \underline{q} & \textit{Set Operation} \\ & \mid & \underline{q} \bowtie_{\underline{\theta}} \underline{q} & \textit{Join} \\ & \mid & \varepsilon & \textit{Empty Relation} \end{array}$$

Figure 2.2: Syntax of relational algebra.

operation, and an empty relation. The syntax is defined in Figure 2.2. The set operations, union and intersection, require two subqueries to have the same relation schema and simply applies the corresponding operation, either union or intersection, to the tuples returned by the subqueries. The *join* operation is equivalent to selection applied to cross-product, that is, $\underline{q}_1 \bowtie_{\underline{\theta}} \underline{q}_2 = \sigma_{\underline{\theta}}(\underline{q}_1 \times \underline{q}_2)$. For example, $\underline{q}_3$ in Example 2.2.1 can be rewritten as

$$\underline{q}_3{}' = (\sigma_{salary \geq 65000}\,(job)) \bowtie_{empacct.title=job.title} empacct \quad .$$

Throughout our examples, omitting the condition of join implies it is a *natural join*, that is, join on the shared attribute of the two subqueries. For example, $\underline{q}_3{}'$ can be rewritten using the natural join

$$\underline{q}_3{}'' = (\sigma_{salary \geq 65000}\,(job)) \bowtie empacct \quad .$$

We also extend relational algebra such that projection of an empty set of attributes is a valid query that returns an empty set of tuples. We define the *empty* query ε as shorthand for projecting an empty set of attributes, that is, $\varepsilon = \pi_{\{\}}\underline{q}$. Note that we do not extend the notation of using underline for relational algebra operators. Instead, relational algebra operators are overloaded and are used as both plain relational and variational operators. It should be clear from context when an operation is variational or not.

Although we do not consider renaming of queries in the formal definition of relational algebra we do support this in our implementation. Furthermore, we use it to rename subqueries of our examples to make them easier to understand. For example, query $\underline{q_3}''$ can be written as:

$$\underline{q_3}'' = \underline{temp} \bowtie empacct$$
$$\underline{temp} \leftarrow \sigma_{salary \geq 65000}\,(job)$$

Making this renaming explicit is necessary to avoid names conflicting in some cases.

2.3 Variation Space and Encoding

In this section, we describe how we represent variation throughout this thesis. We encode variation in terms of *features*. The *feature space*, **FeatName**, of a variational database is a closed set of boolean variables called features. A feature $f \in$ **FeatName** can be enabled (i.e., $f =$ `true`) or disabled ($f =$ `false`). Features

Feature expression syntax:

$$
\begin{array}{rcll}
f \in \textbf{FeatName} & & & \textit{Feature Name} \\
b \in \textbf{Bool} & ::= & \texttt{true} \mid \texttt{false} & \textit{Boolean Value} \\
e \in \textbf{FeatExpr} & ::= & b \mid f \mid \neg e \mid e \wedge e \mid e \vee e & \textit{Feature Expression} \\
c \in \textbf{Config} & : & \textbf{FeatName} \rightarrow \textbf{Bool} & \textit{Configuration}
\end{array}
$$

Evaluation of feature expressions:

$$
\begin{aligned}
\mathbb{E}[\![.]\!] &: \textbf{FeatExpr} \rightarrow \textbf{Config} \rightarrow \textbf{Bool} \\
\mathbb{E}[\![b]\!]_c &= b \\
\mathbb{E}[\![f]\!]_c &= c\ f \\
\mathbb{E}[\![\neg f]\!]_c &= \neg \mathbb{E}[\![f]\!]_c \\
\mathbb{E}[\![e_1 \wedge e_2]\!]_c &= \mathbb{E}[\![e_1]\!]_c \wedge \mathbb{E}[\![e_2]\!]_c \\
\mathbb{E}[\![e_1 \vee e_2]\!]_c &= \mathbb{E}[\![e_1]\!]_c \vee \mathbb{E}[\![e_2]\!]_c
\end{aligned}
$$

Relations over feature expressions:

$$
\begin{aligned}
e_1 \equiv e_2 \ \textit{iff}\ & \forall c \in \textbf{Config}.\mathbb{E}[\![e_1]\!]_c = \mathbb{E}[\![e_2]\!]_c \\
\textit{sat}(e) \ \textit{iff}\ & \exists c \in \textbf{Config}.\mathbb{E}[\![e]\!]_c = \texttt{true} \\
\textit{unsat}(e) \ \textit{iff}\ & \forall c \in \textbf{Config}.\mathbb{E}[\![e]\!]_c = \texttt{false} \\
\textit{oneof}\ (f_1, f_2, \ldots, f_n) = & (f_1 \wedge \neg f_2 \wedge \ldots \wedge \neg f_n) \vee (\neg f_1 \wedge f_2 \wedge \ldots \wedge \neg f_n) \\
& \vee (\neg f_1 \wedge \neg f_2 \wedge \ldots \wedge f_n)
\end{aligned}
$$

Figure 2.3: Feature expression syntax, relations, and evaluation.

describe the variation in a given variational scenario. For example, in the context of schema evolution, features can be generated from version numbers (e.g., features V_1 to V_5 and T_1 to T_5 in the motivating example, Table 1.1); for SPLs, the features can be adopted from the SPL feature set (e.g., the *edu* feature in our motivating example, Table 1.1); and for data integration, the features may represent different data sources. Note that it is easy to extend features to multi-valued variables that have a finite set of countable values. Consider our motivating example, Table 1.1. We could choose to have a feature V as a multi-valued variable which takes one of the values in set $\{1, 2, \ldots, 32\}$. This new multi-valued feature V is equivalent to having five boolean variables V_1, V_2, V_3, V_4, and V_5 since it represents all 32 possible combinations of V_1–V_5.

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 and is denoted by $c \in \mathbf{Config} : \mathbf{FeatName} \rightarrow \mathbf{Bool}$. 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. For example, 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, that is, variant schemas of Table 1.1b. We call a propositional formula of features a *feature expression* and define it formally in Figure 2.3. Additionally, in Figure 2.3, we formally define the evaluation function of feature expressions $\mathbb{E}[\![e]\!]_c$: **FeatExpr → Config → Bool**. This function 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 $\mathbb{E}[\![f_1 \vee f_2]\!]_{\{\}} = \texttt{false} \vee \texttt{false} = \texttt{false}$. Furthermore, in Figure 2.3, we define a binary *equivalence relation* ($\equiv$) relation on feature expressions corresponding to logical equivalence, unary *sat* and *unsat* predicates that determine whether a feature expression is satisfiable or unsatisfiable, respectively. We also define the n-ary operation *oneof* for mutually exclusive features. For example, *oneof* $(V_1, V_2, V_3, V_4, V_5)$ indicates that only one of the features V_1–V_5 can be enabled at a time.

No matter the context, features often have a relationship with each other that constrains the set of possible configurations. For example, in our motivating example (Section 1.1) 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*, which restricts the set of *valid configurations*. That is, a configuration c is only valid if the evaluation of feature model e under configuration c is `true`,

that is $\mathbb{E}[\![e]\!]_c = \texttt{true}$. 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 *oneof* $(V_1, V_2, V_3, V_4, V_5)$.

2.4 Annotations and Variational Sets

We now introduce the first approach used to incorporate variation into a database. 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, that is, if x has type $\mathbf{a}$ (i.e., $x \in \mathbf{a}$) then x^e has the corresponding variational type **Var a** (i.e., $x^e \in$ **Var a**). 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. This is done by the *configuration* function $\mathbf{x}[\![.]\!]$: **Var a** $\rightarrow$ **Config** $\rightarrow \mathbf{a}_\perp$ defined in Figure 2.4. For example, assuming **FeatName** $= \{f_1, f_2\}$, the annotated number $2^{f_1 \vee f_2}$ is present in variants $\{f_1\}$ (i.e., $\mathbf{x}[\![2^{f_1 \vee f_2}]\!]_{\{f_1\}} = 2$), $\{f_2\}$ (i.e., $\mathbf{x}[\![2^{f_1 \vee f_2}]\!]_{\{f_1\}} = 2$), and $\{f_1, f_2\}$ (i.e.,$\mathbf{x}[\![2^{f_1 \vee f_2}]\!]_{\{f_1, f_2\}} = 2$) but not in variant $\{\}$ (i.e., $\mathbf{x}[\![2^{f_1 \vee f_2}]\!]_{\{\}} = \perp$). The operation $pc(x^e) = e$ returns the presence condition of an annotated element.

A *variational set* (*v-set*) $X = \{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}$ is a set of annotated elements, that is, $X \in$ **Set (Var a)** [25, 60, 8]. 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. These plain sets can be generated by

Configuration of an annotated element:

$$\mathbf{x}[\![.]\!] : \mathbf{Var\ a} \rightarrow \mathbf{Config} \rightarrow \mathbf{a}_\perp$$
$$\mathbf{x}[\![x^e]\!]_c = \begin{cases} \underline{x}, & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \perp, & \text{otherwise} \end{cases}$$

Configuration of (annotated) variational sets:

$$\mathbb{X}[\![.]\!] : \mathbf{Set\ (Var\ a)} \rightarrow \mathbf{Config} \rightarrow \mathbf{Set\ a}$$
$$\mathbb{X}[\![\{x_1{}^{e_1}, x_2{}^{e_2}, \ldots, x_n{}^{e_n}\}]\!]_c = \otimes\{\mathbf{x}[\![x_1{}^{e_1}]\!]_c, \mathbf{x}[\![x_2{}^{e_2}]\!]_c, \ldots, \mathbf{x}[\![x_n{}^{e_n}]\!]_c\}$$
$$\mathbb{X}[\![\{\ \}]\!]_c = \{\ \}$$

$$\mathbb{X}[\![.]\!] : \mathbf{Var\ (Set\ (Var\ a))} \rightarrow \mathbf{Config} \rightarrow \mathbf{Set\ a}$$
$$\mathbb{X}[\![X^e]\!]_c = \mathbb{X}[\![\downarrow(X^e)]\!]_c$$
$$\mathbb{X}[\![\{\ \}^{\texttt{true}}]\!]_c = \{\ \}$$
$$\mathbb{X}[\![\{\ \}^{\texttt{false}}]\!]_c = \{\ \}$$

Dropping bots from a plain set:

$$\otimes\{\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_n\} = \{\underline{x}_i \mid \underline{x}_i \neq \perp\}$$

Normalization of (annotated) variational sets:

$$\downarrow(\{x_1{}^{e_1}, x_2{}^{e_2}, \ldots, x_n{}^{e_n}\}) = \{x_i^{e_i} \mid 1 \leq i \leq n, sat(e_i)\}$$
$$\downarrow(X^e) = \{x_i{}^{e_i \wedge e} \mid x_i^{e_i} \in X^e, sat(e_i \wedge e)\}$$

Operations over (annotated) variational sets:

$$X_1 \cup X_2 = \{x^{e_1} \mid x^{e_1} \in\downarrow(X_1), \exists e_2.x^{e_2} \notin\downarrow(X_2)\}$$
$$\cup\{x^{e_2} \mid x^{e_2} \in\downarrow(X_2), \exists e_1.x^{e_1} \notin\downarrow(X_1)\} \cup \{x^{e_1 \vee e_2} \mid x^{e_1} \in\downarrow(X_1), x^{e_2} \in\downarrow(X_2)\}$$
$$X_1 \cap X_2 = \{x^{e_1 \wedge e_2} \mid x^{e_1} \in X_1, x^{e_2} \in X_2, sat(e_1 \wedge e_2)\}$$
$$X_1 \times X_2 = \{(x_1, x_2)^{e_1 \wedge e_2} \mid x_1^{e_1} \in X_1, x_2^{e_2} \in X_2\}$$
$$X_1 \equiv X_2 \textit{ iff } \forall x^e \in (\downarrow(X_1) \cup \downarrow(X_2)), x^{e_1} \in\downarrow(X_1), x^{e_2} \in\downarrow(X_2).e_1 \equiv e_2, e \equiv e_1$$

Figure 2.4: Configuration of variational set and annotated variational set, normalization of variational sets and annotated variational sets, and operations over variational sets. The operations on variational sets are overloaded, that is, $\cup$ is used to both denote the union of variational sets and plain sets.

configuring a variational set with a configuration. This is done by the *variational set configuration* function $\mathbb{X}[\![X]\!]_c$: **Set (Var a) → Config → Set a**, defined in Figure 2.4. The configuration function evaluates the presence condition e_i of each element x_i of the variational set with the configuration c. If the evaluation results in `true` it includes x_i in the plain set and otherwise it does not. Example 2.4.1 illustrates the configuration of a variational set for all possible configurations.

it'd be nice to have the entire ex in the same page.

Example 2.4.1. *Assume we have the feature space* **FeatName** $= \{f_1, f_2\}$ *and the variational set* $X_1 = \{2^{f_1}, 3^{f_2}, 4\}$. X_1 *represents four plain sets:*

$$\mathbb{X}[\![X_1]\!]_c = \begin{cases} \{2,3,4\}, & c = \{f_1, f_2\} \\ \{2,4\}, & c = \{f_1\} \\ \{3,4\}, & c = \{f_2\} \\ \{4\}, & c = \{\} \end{cases}$$

This states that, for example, configuring X_1 *for the variant that enables bot* f_1 *and* f_2 *(that is,* $f_1 =$ `true`, $f_2 =$ `true`*) results in the plain set* $\mathbb{X}[\![X_1]\!]_{\{f_1,f_2\}} = \{2,3,4\}$.

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 variational set*, that is, $X^e \in$ **Var (Set (Var a))**. Annotating a variational set with the feature expression e means that all elements in the variational set are only present when e evaluates to `true`. The *normalization* operation $\downarrow(X^e)$ applies this restriction by pushing it into the presence

conditions of the individual elements: $\downarrow(X^e) = \{x_i{}^{e_i \wedge e} \mid x_i^{e_i} \in X^e, sat(e_i \wedge e)\}$. Note that both the normalization operation and variational set configuration are overloaded, that is, they are defined for both variational sets and annotated variational sets. Also, note that the *normalization* operation also removes elements with unsatisfiable presence conditions and may also be applied to an unannotated variational set X since $X^{\texttt{true}} = X$. For example, the annotated variational 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 variational 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. Note that, without loss of generality, we assume that elements in a variational set are unique since we can simply disjoin the presence conditions of a repeated element, that is, $\{x^e, x^e, x_1^{e_1}, \ldots, x_n^{e_n}\} = \{x^{e \vee e'}, x_1^{e_1}, \ldots, x_n^{e_n}\}$.

In Figure 2.4, we also define several operations, such as union and intersection, over variational sets; these operations are used in Section 4.3. The semantics of a variational set operation is equivalent to applying the corresponding plain set operation to every corresponding variant of the argument variational sets. For example, the union of two variational sets $X_1 \cup X_2$ should produce a new variational set X_3 such that $\forall c \in \textbf{Config}.\ \mathbb{X}[\![X_3]\!]_c = \mathbb{X}[\![X_1]\!]_c \underline{\cup} \mathbb{X}[\![X_2]\!]_c$, where $\underline{\cup}$ is the plain set union operation. This property must hold for all operations over variational sets, that is, for all possible operations, $\odot$, defined on variational sets the property $(P_1) : \forall c \in \textbf{Config}.\mathbb{X}[\![\downarrow(X_1) \odot \downarrow(X_2)]\!]_c = \mathbb{X}[\![X_1]\!]_c \underline{\odot} \mathbb{X}[\![X_2]\!]_c$ must hold, where $\underline{\odot}$ is

the counterpart operation on plain sets.[2]

2.5 The Formula Choice Calculus

The second approach we use to incorporate variation into queries is the formula choice calculus [34] which is an extension of the choice calculus [58, 23]. The choice calculus is a metalanguage for describing variation in programs and its elements such as data structures [60, 25]. In the choice calculus, variation is represented in-place as choices between alternative subexpressions. For example, the variational expression $expr = f_1\langle 1, 2\rangle + f_2\langle 3, 4\rangle + f_1\langle 5, 6\rangle$ contains three choices. Each choice has an associated *dimension*, which is a boolean variable equivalent to a feature and is used to synchronize the choice with other choices in different parts of the expression. For example, expression *expr* contains two dimensions, f_1 and f_2, and the two choices in dimension f_1 are synchronized. Therefore, the variational expression *expr* 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

[2]This property is proved for the operations we define over variational sets in Coq proof assistant [**?**].

and `false` indicates the right alternative, we have:

$$f_1\langle 1,2\rangle + f_2\langle 3,4\rangle + f_1\langle 5,6\rangle = \begin{cases} 1+3+5, & f_1 = \texttt{true}, f_2 = \texttt{true} \\ 1+4+5, & f_1 = \texttt{true}, f_2 = \texttt{false} \\ 2+3+6, & f_1 = \texttt{false}, f_2 = \texttt{true} \\ 2+4+6, & f_1 = \texttt{false}, f_2 = \texttt{false} \end{cases}$$

The formula choice calculus extends the choice calculus by allowing dimensions to be propositional formulas [34]. For example, the variational expression $expr' = f_1 \vee f_2\langle 1,2\rangle + f_2\langle 3,4\rangle + f_1\langle 5,6\rangle$ represents four plain expressions:

$$f_1 \vee f_2\langle 1,2\rangle + f_2\langle 3,4\rangle + f_1\langle 5,6\rangle = \begin{cases} 1+3+5, & f_1 = \texttt{true}, f_2 = \texttt{true} \\ 1+4+5, & f_1 = \texttt{true}, f_2 = \texttt{false} \\ 1+3+6, & f_1 = \texttt{false}, f_2 = \texttt{true} \\ 2+4+6, & f_1 = \texttt{false}, f_2 = \texttt{false} \end{cases}$$

Chapter 3 Variational Database Framework

In this chapter, we introduce the *variational database* framework and how it encodes variation directly in relational databases resulting in a variational database (VDB). To incorporate variation within a database, we annotate elements with feature expressions, as introduced in Section 2.4. We use annotated elements both in the schema and content. Within a schema we allow attributes and relations to exist conditionally based on the feature expression assigned to them (Section 3.1). At the content level, we annotate each tuple with a feature expression, indicating when the tuple is present (Section 3.2).

3.1 Variational Schema

could you pls read this section again up to the paragraph that discusses the formal defition of vdb?

In this section, we define how variation is encoded at the schema level. We first present an example that illustrates ~~why~~ how relational databases fail to encode variation at the schema level and how we can ~~inject~~ express variation in~~to~~ their schemas. Recall from Section 2.1 that the schema of a database is essentially its type. This is also the case for variational databases, except that the components of the schema are variational. The variation in a schema states the condition under which its relations and attributes exist. For example, consider the *empbio* relation schema associated with variant V_3 of our motivating example shown in Table 1.1a. Table 3.1a shows

a corresponding relational table of this relation. Note that the relation *empbio* changes in variants that enable either V_4 or V_5 and Table 3.1b and Table 3.1c show their corresponding relational tables, respectively. The variational relation schema of *empbio* captures this variation in Table 3.1d by annotating the relation and each of its attributes with a feature expression ~~that~~ indicating which configurations they exist in. The feature expressions written in blue above the attributes and the relation name are their presence conditions. For example, the feature express $V_3 \vee V_4 \vee V_5$ indicates that the *empbio* table is present for variants that enable one of V_3–V_5. The three attributes *empno*, *sex*, and *birthdate* ~~exists in the relation *empbio* no matter what as long as *empbio* exists in a variant~~ are present in all variants where the empbio relation exists, so their presence condition are `true`. However, the *name* attribute is only present in variants that enable V_4 while attributes *firstname* and *lastname* are only present in variants that enable V_5.

Furthermore, the existence of the variational relation *empbio* and its attributes relies on the existence of the entire variational database which is captured by the feature model of the database. Remember that the feature model is the presence condition of the variational database as a whole. The hierarchy of presence condition sometimes simplifies the presence conditions. For example, assuming that only one of the V_3–V_5 (missing space) can be enabled for a variant at a time, the *empbio* variational table shown in Table 3.1d can be simplified to the variational table shown in Table 3.2. Note that Table 3.2 and Table 3.1d focus~~e~~ only on the relation schema and do~~es~~ not include the variation at the content level.[1] We discuss the encoding of variation at the content level in Section 3.2.

[1] Table 3.4 includes the variation at the content level.

Table 3.1: The relational tables of *empbio* for variants that enable one of the features V_3, V_4, or V_5 and the variational relation *empbio* that encompasses the three variants of the plain table *empbio* without accounting for variation at the content level.

(a) The relational table of *empbio* for variants that only enable the feature V_3 out of the features V_1–V_5. The relation schema is captured by the name of the relation and its attributes.

empbio

empno	*sex*	*birthdate*
12001	F	1960-11-06
12002	M	1961-04-15
12003	M	1958-07-27
...	...	...

(b) The relational table of *empbio* for variants that only enable the feature V_4 out of the features V_1–V_5. The relation schema is captured by the name of the relation and its attributes.

empbio

empno	*sex*	*birthdate*	*name*
80001	M	1956-09-30	Nagui Merli
80002	M	1963-04-25	Mayuko Meszaros
80003	F	1960-10-26	Theirry Viele
...	...	...	...

(c) The relational table of *empbio* for variants that only enable the feature V_5 out of the features V_1–V_5. The relation schema is captured by the name of the relation and its attributes.

empbio

empno	*sex*	*birthdate*	*firstname*	*lastname*
200000	M	1960-01-11	Selwyn	Koshiba
200001	M	1957-09-10	Bedrich	Markovitch
200002	F	1961-02-07	Pascal	Benzmuller
...	...	...	...	...

(d) The variational relation of *empbio* without accounting for variation at the content level. The relation schema is captured by the name of the relation and attributes in addition to their presence conditions which are colored blue.

empbio

$V_3 \vee V_4 \vee V_5$	true	true	true	$V_4 \wedge \neg V_3 \wedge \neg V_5$	$V_5 \wedge \neg V_3 \wedge \neg V_4$	$V_5 \wedge \neg V_3 \wedge \neg V_4$
empbio	*empno*	*sex*	*birthdate*	*name*	*firstname*	*lastname*
	12001	F	1960-11-06			
	12002	M	1961-04-15			
	12003	M	1958-07-27			
	80001	M	1956-09-30	Nagui Merli		
	80002	M	1963-04-25	Mayuko Meszaros		
	80003	F	1960-10-26	Theirry Viele		
	200001	M	1960-01-11		Selwyn	Koshiba
	200002	M	1957-09-10		Bedrich	Markovitch
	200003	F	1961-02-07		Pascal	Benzmuller
	...	...	...	...	...	...

Table 3.2: The variational relation *empbio* without accounting for variation at the content level. This table is present under a presence condition e_{mot} that applies to the entire database of our motivating example. Example 3.1.2 provides the variational schema of our motivating example and explains e_{mot}.

$V_3 \vee V_4 \vee V_5$	true	true	true	V_4	V_5	V_5
empbio	*empno*	*sex*	*birthdate*	*name*	*firstname*	*lastname*
	12001	F	1960-11-06			
	12002	M	1961-04-15			
	12003	M	1958-07-27			
	80001	M	1956-09-30	Nagui Merli		
	80002	M	1963-04-25	Mayuko Meszaros		
	80003	F	1960-10-26	Theirry Viele		
	200001	M	1960-01-11		Selwyn	Koshiba
	200002	M	1957-09-10		Bedrich	Markovitch
	200003	F	1961-02-07		Pascal	Benzmuller
	...	...	...	...	...	...

Figure 3.1 gives a formal definition of variational schemas. Variational schemas build on plain relational schemas defined in Section 2.1. A variational schema captures variation in the structure of a database by indicating which attributes and relations are included or excluded in which variants. To this end, we annotate attributes, relations, and the schema itself with feature expressions, which describe the conditions under which each is present. A *variational relation schema* (*v-relation schema*), s, is a relation name with an annotated variational set of attributes, $s \in \mathbf{RelSch} \coloneqq r\,(A)^e$. The presence condition of the variational relation schema, e, determines in what variants of the database the relation itself is present. A *variational schema* (*v-schema*) is an annotated set of variational relation schemas, $S \in \mathbf{Sch} \coloneqq \{s_1, \ldots, s_n\}^e$. The presence condition of the entire variational schema, e, is the VDB's feature model, which provides a top-level constraint on the set of valid configurations, as described in Section 2.3, and can

be extracted by $pc(S)$. Hence, the variational schema defines all valid schema variants of a VDB. Example 3.1.1 defines a variational schema for only a part of our motivating example introduced in Section 1.1 and Example 3.1.2 provides the variational schema of our motivating example in its entirety.

Example 3.1.1. *S_1 is the variational schema of a VDB that only includes relations empacct and ecourse in the last two rows of both Table 1.1a and Table 1.1b. It has the feature space* $\mathbf{FeatName} = \{V_4, V_5, edu, T_4, T_5\}$. *Note that attributes that exist conditionally are annotated with a feature expression to account for such a condition, e.g., the salary attribute only exists when* $V_5 =$ `true`.

$$S_1 = \{empacct(empno, hiredate, title, deptno, salary^{V_5}, std^{edu}, instr^{edu})^{V_4 \vee V_5}$$
$$, ecourse(courseno, coursename, deptno^{T_5})^{T_4 \vee T_5}\}^{e_1}$$
$$e_1 = (\neg edu \wedge oneof\ (V_4, V_5) \wedge \neg\ (T_4 \vee T_5)) \vee (edu \wedge oneof\ (V_4, V_5) \wedge oneof\ (T_4, T_5))$$

where e_1 allows only one temporal feature for the basic schema to be enabled at a time, and either one temporal feature for the education extension, if edu is enabled, or else no temporal feature for the education extension.

The presence of an attribute follows the hierarchal layout of information in a database: an attribute's presence depends on the presence of its parent variational relation, which in turn depends on the presence of the variational schema. Thus, the complete presence condition of the attribute a^{e_a} in variational relation $r(\ldots)^{e_r}$ defined in variational schema S is $pc(a, S) = e_a \wedge e_r \wedge pc(S)$.

Similarly, the presence condition of variational relation r is $pc(r, S) = e_r \wedge pc(S)$.

Variational schema objects:

$$
\begin{array}{rcll}
a \in \textbf{AttrName} & & & \textit{Attribute Name} \\
r \in \textbf{RelName} & & & \textit{Relation Name} \\
A \in \textbf{Set (Var AttrName)} & \coloneqq & \{a_1^{e_1}, a_2^{e_2}, \ldots, a_k^{e_k}\} & \textit{Variational Set of Attributes} \\
s \in \textbf{RelSch} & \coloneqq & r\,(A)^e & \textit{Variational Relation Schema} \\
S \in \textbf{Sch} & \coloneqq & \{s_1, \ldots, s_n\}^e & \textit{Variational Schema}
\end{array}
$$

Variational schema type synonyms:

$$\textbf{Sch} = \textbf{Var (Set RelSch)}$$

Presence condition of attributes and relations:

$$pc(a, S) = pc(a, r(\ldots, a^{e_a}, \ldots)^{e_r}) \wedge pc(r, S) = pc(a^{e_a}) \wedge pc(r, S) = e_a \wedge e_r \wedge pc(S)$$
$$pc(r, S) = pc(r(A)^{e_r}) \wedge pc(S) = e_r \wedge pc(S)$$

Variational set of attributes configuration:

$$\mathbb{A}[\![.]\!] : \textbf{Set (Var AttrName)} \rightarrow \textbf{Config} \rightarrow \textbf{AttrName}$$
$$\mathbb{A}[\![A]\!]_c = \mathbb{X}[\![A]\!]_c$$

Variational relation schema configuration:

$$\mathbb{R}[\![.]\!] : \textbf{RelSch} \rightarrow \textbf{Config} \rightarrow \underline{\textbf{RelSch}}_{\perp}$$
$$\mathbb{R}[\![r\,(A)^{e_A}]\!]_c = \begin{cases} r\,(\mathbb{A}[\![\downarrow(A^{e_A})]\!]_c)\,, & \text{if } \mathbb{E}[\![e_A]\!]_c = \texttt{true} \\ \perp, & \text{otherwise} \end{cases}$$

Variational schema configuration:

$$\mathbb{S}[\![.]\!] : \textbf{Sch} \rightarrow \textbf{Config} \rightarrow \underline{\textbf{Sch}}$$
$$\mathbb{S}[\![\{r_1\,(A_1)^{e_1}, \ldots, r_n\,(A_n)^{e_n}\}^e]\!]_c = \begin{cases} \{\mathbb{R}[\![r_1(A_1)^{e_1 \wedge e}]\!]_c, \ldots, \mathbb{R}[\![r_n(A_n)^{e_n \wedge e}]\!]_c\}, & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \{\}, & \text{otherwise} \end{cases}$$

Figure 3.1: Variational schema definition, presence condition of attributes and relations, and variational schema configuration.

For example, in Example 3.1.1 we have $pc(empacct, S_1) = (V_4 \vee V_5) \wedge e_1$. Furthermore, a database element is only present in variants for which its presence condition satisfiable, e.g., in Example 3.1.1 the *std* attribute is present in the variant $\{edu, V_4, T_5\}$ since $\mathbb{E}[\![pc(std, S_1)]\!]_{\{edu,V_4,T_5\}} = \mathbb{E}[\![edu \wedge (V_4 \vee V_5) \wedge e_1]\!]_{\{edu,V_4,T_5\}} =$ $\texttt{true} \wedge (\texttt{true} \vee \texttt{false}) \wedge ((\neg\texttt{true} \wedge \mathit{oneof}\ (\texttt{true}, \texttt{false}) \wedge \neg(\texttt{false} \vee \texttt{true})) \vee (\texttt{true} \wedge \mathit{oneof}\ (\texttt{true}, \texttt{false}) \wedge \mathit{oneof}\ (\texttt{false}, \texttt{true}))) = \texttt{true}$ but it is not present in the variant $\{V_4, T_5\}$ since in this variant $edu = \texttt{false}$, thus, $\mathbb{E}[\![pc(std, S_1)]\!]_{\{edu,V_4,T_5\}} = \texttt{false}$.

Intuitively and similar to variational sets, a variational schema is a systematic compact representation of a set of plain schemas called variants. A schema variant can be obtained by *configuring* the variational schema with that variant's configuration. We define the configuration function for variational schemas and its elements in Figure 3.1. Example 3.1.2 illustrates configuring the variational schema of our motivating example for the variant $\{edu, V_2, T_3\}$.

Example 3.1.2. *Table 3.3 illustrates the variational schema of the motivating example, denoted by S_{mot}. As a reminder the motivating example has the feature space* **FeatName** $= \{edu, V_1, V_2, V_3, V_4, V_5, T_1, T_2, T_3, T_4, T_5\}$. *Additionally, all schema variants are illustrated inTable 1.1.*

make sure encoding is correct in Table 3.3

The feature model e_{mot} only allows one temporal feature to be true from a set

Table 3.3: Variational schema S_{mot} with feature model e_{mot}. This variational schema encompasses 30 relational schemas: five schemas when edu = `false` and 25 schemas otherwise.

$engineerpersonnel(empno, name, hiredate, title, deptname)^{V_1}$
$otherpersonnel(empno, name, hiredate, title, deptname)^{V_1}$
$empacct(empno, name^{V_2 \vee V_3}, hiredate, title, deptname^{V_2}, deptno^{V_3 \vee V_4 \vee V_5}, salary^{V_5},$ $std^{V_4 \vee V_5}, instr^{V_4 \vee V_5})^{V_2 \vee V_3 \vee V_4 V_5}$
$job\,(title, salary)^{V_1 \vee V_2 \vee V_3 \vee V_4}$
$dept\,(deptname, deptno, managerno, stdnum^{V_5}, instrnum^{V_5})^{V_3 \vee V_4 \vee V_5}$
$empbio\,(empno, sex, birthdate, name^{V_4}, firstname^{V_5}, lastname^{V_5})^{V_3 \vee V_4 \vee V_5}$
$course\,(courseno^{\neg T_1}, coursename, teacherno^{T_1 \vee T_2}, time^{T_4 \vee T_5}, class^{T_4 \vee T_5}, deptno^{T_5})^{edu}$
$student\,(studentno, coursename^{T_1}, courseno^{\neg T_1}, grade^{T_3 \vee T_4})^{edu \wedge \neg T_5}$
$teach\,(teacherno, courseno)^{edu \wedge (T_3 \vee T_4 \vee T_5)}$
$ecourse\,(courseno, coursename, deptno^{T_5})^{edu \wedge (T_4 \vee T_5)}$
$take\,(studentno, courseno, grade)^{edu \wedge T_5}$

of temporal features at the time.

$$e_{mot} = (\neg edu \wedge oneof\,(V_1, V_2, V_3, V_4, V_5) \wedge \neg\,(T_1 \vee T_2 \vee T_3 \vee T_4 \vee T_5)) \\ \vee\,(edu \wedge oneof\,(T_1, T_2, T_3, T_4, T_5) \wedge oneof\,(V_1, V_2, V_3, V_4, V_5))$$

However, the feature model can be encoded differently. For example, e'_{mot} restrict it such that it only allows the two sets of temporal features to change together.

$$e'_{mot} = oneof((V_1 \vee (V_1 \wedge edu \wedge T_1))\,, (V_2 \vee (V_2 \wedge edu \wedge T_2))\,, (V_3 \vee (V_3 \wedge edu \wedge T_3)) \\ ,\,(V_4 \vee (V_4 \wedge edu \wedge T_4))\,, (V_5 \vee (V_5 \wedge edu \wedge T_5)))$$

Hence, the feature model of a VDB can vary based on the relationship between

features and the restrictions that they must follow. Additionally, the encoding of presence condition can change since different feature expressions can be indicate the same set of variants. For example, the presence condition of the job relation can be changed to $\neg V_5$.

Configuring the variational schema S_{mot} *for the variant that only enables features edu,* V_2*, and* T_3 *(i.e., the variant* $\{edu, V_2, T_3\}$*) results in relations contained in the two yellow highlighted cells of tables in Table 1.1.*

3.2 Variational Table

Thus far, we illustrated how variation can be incorporated into the schema of a database. However, it does not suffice to only have variation at the schema level. Consider the variants of the relation *empbio* shown in Table 3.1a–Table 3.1c. Table 3.2 illustrates the variational relation *empbio* but it does not include variation at the content level. To incorporate variation in the content of a variational relation we extend each relation with a new attribute *prescond* that stores the presence condition of tuples, as shown in Table 3.4. Thus, the value of this attribute for a tuple determines the set of variants that the tuple belongs to. For example, the first tuple in the Table 3.4 is present for all database variants that enable the feature V_3. Note that the white spaces in Table 3.4 indicate the non-existing values for an attribute in a tuple. For example, consider the first three tuples of the variational table *empbio* shown in Table 3.4. Since none of the attributes *name*, *firstname*, and *lastname* exists for variants that enable V_3 they are left empty for tuples of

Table 3.4: The variational table of *empbio* encompassing the three variants of the plain relation *empbio* shown in Table 3.1a–Table 3.1c. This table accounts for both variation at the schema and content levels. Note that feature model e_{mot} of the entire VDB applies to it.

$V_3 \vee V_4 \vee V_5$	true	true	true	V_4	V_5	V_5	true
empbio	*empno*	*sex*	*birthdate*	*name*	*firstname*	*lastname*	*prescond*
	12001	F	1960-11-06				V_3
	12002	M	1961-04-15				V_3
	12003	M	1958-07-27				V_3
	80001	M	1956-09-30	Nagui Merli			V_4
	80002	M	1963-04-25	Mayuko Meszaros			V_4
	80003	F	1960-10-26	Theirry Viele			V_4
	200001	M	1960-01-11		Selwyn	Koshiba	V_5
	200002	M	1957-09-10		Bedrich	Markovitch	V_5
	200003	F	1961-02-07		Pascal	Benzmuller	V_5
	...	...	...	...	...	...	...

variants that enable V_3.

To account for content variability in the formal definition we tag tuples with presence conditions. Thus, a *variational tuple* (*v-tuple*) is an annotated tuple, $u \in$ **Tuple** $\coloneqq (v_1, \ldots, v_l)^e$. A variational tuple corresponds to a variational relation, $r(a_1, \ldots, a_l)^{e_r}$, where each element v_i is a value corresponding to attribute a_i (recall that attributes in a variational relation are ordered). For example, $(38, PL, 678)^{T_5}$ is a variational tuple that belongs to the *ecourse* relation from Example 3.1.1 and is only present when T_5 is enabled. The content of a variational relation is a set of variational tuples, $U \in$ **RelCont** $\coloneqq \{u_1, \ldots, u_k\}$ and a *variational table* (*v-table*) is the pair of its relation schema and content, $t = (s, U)$. A *variational database instance* is a set of variational tables, $\mathcal{I} \in$ **DBInst** $\coloneqq \{t_1, \ldots, t_n\}^e$. Figure 3.2 provides the formal definition of a VDB and its type synonyms. A VDB instance is *well-formed* if its encoded variation at the schema and content level are consistent and satisfiable [10]. Section 3.3 expands on this.

Variational database objects:

$$
\begin{array}{rcll}
u \in \textbf{Var Tuple} & \coloneqq & (v_1, \ldots, v_l)^e & \textit{Variational Tuple} \\
U \in \textbf{RelCont} & \coloneqq & \{u_1, \ldots, u_k\} & \textit{Variational Relation Content} \\
t \in \textbf{Table} & \coloneqq & (s, U) & \textit{Variational Table} \\
\mathcal{I} \in \textbf{DBInst} & \coloneqq & \{t_1, \ldots, t_n\}^e & \textit{Variational Database Instance}
\end{array}
$$

Variational database type synonyms:

$$
\begin{aligned}
\textbf{RelCont} &= \textbf{Set}\ (\textbf{Var Tuple}) \\
\textbf{Table} &= (\textbf{RelSch}, \textbf{RelCont}) \\
\textbf{DBInst} &= \textbf{Var}\ (\textbf{Set}\ ((\textbf{RelSch}, \textbf{RelCont})))
\end{aligned}
$$

Variational tuple configuration:

$$
\begin{aligned}
&\mathbb{U}[\![.]\!] : \textbf{Var Tuple} \to \textbf{RelSch} \to \textbf{Config} \to \textbf{Tuple}_\perp \\
&\mathbb{U}[\![(v_1, \ldots, v_l)^e]\!]_{(s,c)} \\
&\qquad = \begin{cases} (v_i, \cdots, v_j), & \text{if } \forall k. 1 \leq i \leq k \leq j \leq l, \mathbb{E}[\![pc(att(k), s) \wedge e]\!]_c = \texttt{true} \\ \perp, & \text{otherwise} \end{cases}
\end{aligned}
$$

Variational relation content configuration:

$$
\begin{aligned}
&\mathbb{T}[\![.]\!] : \textbf{RelCont} \to \textbf{RelSch} \to \textbf{Config} \to \underline{\textbf{RelCont}} \\
&\mathbb{T}[\![\{u_1, \ldots, u_k\}]\!]_{(s,c)} = \{\mathbb{U}[\![u_1]\!]_{(s,c)}, \ldots, \mathbb{U}[\![u_k]\!]_{(s,c)}\}
\end{aligned}
$$

VDB instance configuration:

$$
\begin{aligned}
&\mathbb{I}[\![.]\!] : \textbf{DBInst} \to \textbf{Config} \to \underline{\textbf{DBInst}} \\
&\mathbb{I}[\![\{t_1, \ldots, t_n\}^e]\!]_c = \mathbb{I}[\![\{(s_1, U_1), \ldots, (s_n, U_n)\}^e]\!]_c \\
&\qquad = \begin{cases} \{\left(\mathbb{R}[\![r_1(A_1)^{e_1 \wedge e}]\!]_c, \mathbb{T}[\![\downarrow \left(U_1^{e_1 \wedge e}\right)]\!]_{(s_1,c)}\right), \ldots\}, & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \{\}, \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.2: VDB instance syntax and configuration. Note that the schema of a relation must be passed to the configuration function for its content, however, the variational schema does not need to be passed to configuration functions of smaller parts of the variational schema such as $\mathbb{R}[\![.]\!]_c$ or $\mathbb{A}[\![.]\!]_c$ since all needed information for configuring a part of a variational schema is propagated.

Similar to a variational schema, a user can configure a variational table or a VDB for a specific variant, formally defined in Figure 3.2. This allows users to deploy a VDB for a specific configuration and generate the corresponding VDB variant. For example, configuring the variational table *empbio*, shown in Table 3.4, for configuration $\{V_3, edu, T_1\}$ results in the relational table *empbio* in Table 3.1a, assuming the VDB has the variational schema S_{mot} given in Example 3.1.2. Additionally, our VDB framework puts all variants of a database into one VDB and it keep tracks of which variant a tuple belongs to by annotating them with presence conditions. For example, consider tuples $(38, PL, 678)^{T_5}$ and $(23, DB, \texttt{NULL})^{T_4}$ that belong to the *ecourse* table. The presence conditions T_5 and T_4 state that tuples belong to temporal variants four and five of this VDB, respectively. Hence, this framework tracks which variants a tuple belongs to.

Our VDB framework encodes variation in databases at two levels: schema and content. In a variational database, while content-level variation can stand on its own, such as frameworks used for database versioning and experimental databases [33], the schema level cannot. For example, $ecourse\left(courseno, coursename, deptno^{T_5}\right)^{ed}$ encodes variation at the schema level for relation *ecourse*. Dropping the presence conditions of tuples leads to ambiguity, i.e., it is unclear which variant each of the tuples $(38, PL, 678)$ and $(23, DB, \texttt{NULL})$ belongs to. We can only guess that they belong to variants where T_4 or T_5 are enabled, but, we do not know for sure which one.

if time allows give this example over the Table 3.4.

Note that we limit the granularity of variation in content to tuples, that is, the individual values within a tuple are not variational. This design deci-

sion causes some redundancy. For example, the two tuples $(38, PL, 678)^{T_5}$ and $(38, PL, \texttt{NULL})^{\neg T_5}$ cannot be represented as a single tuple with variation in the third element. However, this design decision does not prevent us from distinguishing between a `NULL` value that represents a missing value and a `NULL` value that represents a cell that is not present. This distinction can be made by checking the satisfiability of the presence condition of the value v_i in tuple u with of relation r in schema S: If $sat(pc(att(i), S) \wedge pc(u, r) \wedge pc(r, S))$ then the `NULL` indicates a missing value and otherwise it indicates a non-present cell.

3.3 Properties of a Variational Database Framework

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 VDB feature model $pc(S)$:

$$sat(pc(S))$$

2. Every relation r is present in at least one configuration of the variational schema S:

$$\forall r \in S, sat(pc(r, S))$$

3. Every attribute a in every relation r is present in at least one configuration of the variational schema S:

$$\forall a \in r, \forall r \in S.sat(pc(r, S) \wedge pc(a, r))$$

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:

$$\forall c \in \textbf{Config}.\mathbb{S}[\![S]\!]_c = S_c$$

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:

$$\forall u \in r, \forall r \in S.sat(pc(r, S) \wedge pc(u, r))$$

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:

$$\forall u \in r, a \in r, r \in S.\neg sat(pc(r, S) \wedge pc(a, r) \wedge pc(u, r)) \Rightarrow value_u(a) = \text{NULL}$$

Since a single VDB can supply data for many different database variants at the same time, encoding variation explicitly in a database allows the developers to check for different properties over all database variants. Thus, depending on the context of the VDB, more specialized properties can be checked. 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, introduced in Section 5.2. To check this, assume that configurations $c_1, c_2, \cdots$ represent time-ordered configurations, then check $\forall c_i, c_j \in \mathbf{Config}, i \leq j, \mathbb{I}[\![\mathcal{I}]\!]_{c_i} \subseteq \mathbb{I}[\![\mathcal{I}]\!]_{c_j}$, where $\mathbb{I}[\![\mathcal{I}]\!]_c$ denotes configuring the VDB instance $\mathcal{I}$ for configuration c, defined in Figure 3.2.

Chapter 4 Variational Queries

change all $V_4 \vee V_5$ to $\neg V_3$ in and update the derivation tree examples to the final type system.

pls read the following paragraph

Now that we have introduced the variational database framework we need a query language to extract information from a VDB instance ~~and unlike~~ relational query languages (like SQL and relational algebra) ~~it needs to~~ account for the new aspect of our database: variation.

. Our approach will build on existing but must also

We formally define *variational relational algebra* (VRA) in Section 4.1 as our algebraic query language. A query written in VRA is called a *variational query* (*v-query*); we use query and variational query interchangeably when it is clear from context. Unlike relational queries that convey an intent over a single database, a variational query typically conveys the same intent over several relational database variants. However, a single variational query is also capable of capturing different intents over different database variants.

pls read the following paragraph

To understand the meaning of variational queries we define the semantics of variational queries via the semantics of relational queries in Section 4.2. We define how ~~approaches~~ to configure a variational query to a relational query in Section 4.2.1. Then~~,~~ we use the results of multiple relational queries to accumulate the result of the original variational query in Section 4.2.2.

Due to the expressiveness of variational queries, we define a type system for VRA that statically checks a variational query against the underlying variational schema in Section 4.3. To make variational queries more useable we relieve the

user from repeating the variational schema's variation in their variational queries. This is achieved by explicitly annotating queries in Section 4.4. We then define the *variation-preservation* property for VRA at the type level in Section 4.6.2. Finally, we provide a set of syntactic rules that are semantic-preserving in Section 4.5 that enable factoring and distributing variation points within a variational query, which enables syntactic refactorings including maximizing sharing within a variational query.

4.1 Variational Relational Algebra

To account for variation, VRA combines relational algebra (RA) with *choices* [23, 34, 58]. Remember that a choice $e\langle x_1, x_2\rangle$ consists of a feature expression e, called the *dimension* of the choice, and two *alternatives* x_1 and x_2. For a given configuration c, the choice $e\langle x_1, x_2\rangle$ can be replaced by x_1 if e evaluates to `true` under configuration c, (i.e., $\mathbb{E}[\![e]\!]_c$), or x_2 otherwise.

The syntax of VRA is given in Figure 4.1. The selection operation is similar to standard RA selection except that the condition parameter is *variational* meaning that it may contain choices. For example, the query $\sigma_{e\langle a_1=a_2, a_1=a_3\rangle}(r)$ selects a variational tuple u if it satisfies the condition $a_1 = a_2$ and $sat(e \wedge pc(u))$ or if $a_1 = a_3$ and $sat(\neg e \wedge pc(u))$. The projection operation is parameterized by a variational set of attributes, A. For example, the query $\pi_{a_1, a_2{}^e}(r)$ projects a_1 from relation r unconditionally, and a_2 when $sat(e)$. The choice operation enables combining two variational queries to be used in different variants based on the dimension. In

Operators:

$$\begin{array}{rcl} \bullet & ::= & < \mid \leq \mid = \mid \neq \mid > \mid \geq \\ \circ & ::= & \cup \mid \cap \end{array}$$

Variational conditions:

$$\begin{array}{rcl} \theta \in \textbf{Condition} & ::= & b \mid a \bullet k \mid a \bullet a \mid \neg\theta \mid \theta \vee \theta \\ & \mid & \theta \wedge \theta \mid e\langle\theta, \theta\rangle \end{array}$$

Variational queries:

$$\begin{array}{rcll} q \in \mathbf{Q} & ::= & r & \textit{Relation} \\ & \mid & \sigma_\theta q & \textit{Selection} \\ & \mid & \pi_A q & \textit{Projection} \\ & \mid & e\langle q, q\rangle & \textit{Choice} \\ & \mid & q \times q & \textit{Cartesian Product} \\ & \mid & q \circ q & \textit{Set Operation} \\ & \mid & \varepsilon & \textit{Empty Relation} \end{array}$$

Figure 4.1: Syntax of variational relational algebra.

practice, it is often useful to return information in some variants and nothing at all in others. We introduce an explicit *empty* query ε to facilitate this. Similar to our definition of the empty query for relational algebra, for VRA we also have: $\varepsilon = \pi_{\{\}} q$. The empty query is used, for example, in q_2 in Example 4.1.1. The rest of VRA's operations are similar to RA, where all set operations (union, intersection, and product) are changed to the corresponding variational set operations defined in Section 2.4.

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

in this thesis.

The result of a variational query is a variational table with the reserved relation name *result*. For example, assume that variational tuples $(1,2)^{f_1}$ and $(3,4)^{\neg f_3}$ belong to a variational relation $r(a_1, a_2)$, which is the only relation in a VDB with the trivial feature model `true`. The query $f_3\langle\pi_{a_1{}^{f_2}}(r), \varepsilon\rangle$ returns a variational table with relation schema $result(a_1{}^{f_2})^{f_3}$, which indicates that the result is only non-empty when f_3 is `true` and that the result includes attribute a_1 when f_2 is `true`. The content of the result relation for the example query is a single variational tuple $(1)^{f_1}$. The tuple $(3)^{\neg f_3}$ is not included since the projection occurs in the context of a choice in f_3, which is incompatible with the presence condition of the tuple, i.e., $unsat(f_3 \wedge \neg f_3)$. This illustrates how choices can effectively filter the tuples in a VDB based on the dimension. Example 4.1.1 illustrates how a variational query can be used to express variational information needs.

Example 4.1.1. *Assume a VDB with* **FeatName** $= \{V_3, V_4, V_5\}$*, and the only variational table empbio shown in Table 3.4. The VDB has the feature model* $e_2 = oneof\,(V_3, V_4, V_5)$ *which states that the three* V_3–V_5 *are mutually exclusive. Note that* e_2 *is different from the feature model* e_{mot} *of the empbio variational table shown in Table 3.4 . The variational schema for this VDB is:*

pls read this example. it has changed a bit

$$S_2 = \{empbio(empno, sex, birthdate, name^{V_4}, firstname^{V_5}, lastname^{V_5})\}^{e_2}$$

Now, the user wants the employee ID numbers (empno) and their names for variants that enable either V_4 *or* V_5 *but not* V_3*. We show the steps to build up*

multiple queries that can extract this information. First, to extract the required attributes we write the query q_0 to project all the needed attributes without considering the variational aspect of projection. Note that the presence condition attribute (prescond) does not need to be projected. In fact, the presence condition attribute is returned for every variational query since that is the only way to keep track of variation at the content level. Table 4.1a shows the result of this query over the described VDB. Note that the presence condition of the result is $pc(empbio, S_2) = oneof\ (V_3, V_4, V_5) \wedge (V_3 \vee V_4 \vee V_5)$ which can be simplified to $oneof\ (V_3, V_4, V_5)$. We discuss how the presence conditions of the returned result and its attributes are generated in Section 4.3.

(show q0 here)

(show q0 here)

$$q_0 = \pi_{empno,name,firstname,lastname}(empbio)$$

(create a new paragraph here after moving q0 up)

Now we pay attention to the variational aspect of the query. Knowing that the database enforces the variation encoded in itself (that is, the VDB exists if and only if exactly one of the features V_3–V_5 is enabled, the name attribute only exists for variants that enable V_4 and the firstname and lastname attributes only exist for variants that enable V_5) and since we only want the projected attributes for variants that enable V_4 or V_5 we can write the query q_1. Table 4.1b shows the result of this query over the described VDB. Note that the first three tuples from Table 4.1a are not returned since the empno attribute does not exist for them in addition to the previously non-existing attributes name, firstname, and lastname. Thus, the tuple

(show q1 here)

could just say here, "if desired, we can also make the inferred presence conditions explicit, as demonstrated in the following query q1'."

will just be empty and so is dropped.

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

If we did not know that the database enforces the variation encoded in itself we had to repeat that variation. This is expressed in q_1'. *The result of this query is still Table 4.1b.*

$$q_1' = \pi_{empno^{(V_4 \vee V_5) \wedge \neg V_3}, name^{V_4 \wedge \neg V_3 \wedge \neg V_5}, firstname^{V_5 \wedge \neg V_3 \wedge \neg V_4}, lastname^{V_5 \wedge \neg V_3 \wedge \neg V_4}}(empbio)$$

Note that unlike the variational table empbio shown in Table 3.4, the result of a variational query is not part of a bigger variational structure (the VDB). Thus, it is only restricted by its presence condition and not the feature model although the feature model has already been applied to its presence condition.

isn't this a moot point? since the feature model has already been applied, the result is unchanged by whether the feature model restricts it or not?

In the example, note that the user does not need to repeat the variability encoded in the variational schema in their query, that is, they do not need to annotate *name*, *firstname*, and *lastname* with V_4, V_5, and V_5, respectively. We discuss this in more detail in Section 4.4. q_1 queries all three variants simultaneously although the returned results are only associated with variants V_4 and V_5 due to the annotation of the attribute *empno* in the query and the presence conditions of the rest of the projected attributes in the schema. Yet, the query can be ~~more~~ further simplified with a choice. q_2 selects only two out of the three variants explicitly: $q_2 = \neg V_3 \langle \pi_{empno, name, firstname, lastname}(empbio), \varepsilon \rangle$. Table 4.1c shows the result fo this

Table 4.1: Results of some variational queries over the VDB instance described in Example 4.1.1.

(a) Result of the variational query $\pi_{empno,name,firstname,lastname}(empbio)$.

oneof (V_3, V_4, V_5)	`true`	V_4	V_5	V_5	`true`
result	*empno*	*name*	*firstname*	*lastname*	*prescond*
	12001				V_3
	12002				V_3
	12003				V_3
	80001	Nagui Merli			V_4
	80002	Mayuko Meszaros			V_4
	80003	Theirry Viele			V_4
	200001		Selwyn	Koshiba	V_5
	200002		Bedrich	Markovitch	V_5
	200003		Pascal	Benzmuller	V_5
	...	...	...	...	...

(b) Result of the variational queries $\pi_{empno^{V_4 \vee V_5},name,firstname,lastname}(empbio)$ and $\pi_{empno^{(V_4 \vee V_5) \wedge \neg V_3},name^{V_4 \wedge \neg V_3 \wedge \neg V_5},firstname^{V_5 \wedge \neg V_3 \wedge \neg V_4},lastname^{V_5 \wedge \neg V_3 \wedge \neg V_4}}(empbio)$.

oneof (V_3, V_4, V_5)	$V_4 \vee V_5$	V_4	V_5	V_5	`true`
result	*empno*	*name*	*firstname*	*lastname*	*prescond*
	80001	Nagui Merli			V_4
	80002	Mayuko Meszaros			V_4
	80003	Theirry Viele			V_4
	200001		Selwyn	Koshiba	V_5
	200002		Bedrich	Markovitch	V_5
	200003		Pascal	Benzmuller	V_5
	...	...	...	...	...

(c) Result of the variational query $\neg V_3 \langle \pi_{empno,name,firstname,lastname}(empbio), \varepsilon \rangle$.

oneof $(V_3, V_4, V_5) \wedge \neg V_3$	`true`	V_4	V_5	V_5	`true`
result	*empno*	*name*	*firstname*	*lastname*	*prescond*
	80001	Nagui Merli			V_4
	80002	Mayuko Meszaros			V_4
	80003	Theirry Viele			V_4
	200001		Selwyn	Koshiba	V_5
	200002		Bedrich	Markovitch	V_5
	200003		Pascal	Benzmuller	V_5
	...	...	...	...	...

query over the VDB described in Example 4.1.1. Note that, as shown in Table 4.1, queries q_1 and q_2 return the same set of variational tuples since neither returns tuples associated with variant V_3, but their returned variational tables have different presence conditions, thus, q_2 filters out tuples that belong to variant V_3 at the schema level while q_1 does not. We discuss this more in Example **??**.

Expressing the same intent over several database variants by a single query relieves the DBA from maintaining separate queries for different variants or configurations of the schema. Example 4.1.2 illustrates this point.

Example 4.1.2. *Assume a VDB with* **FeatName** $= \{V_1, \ldots, V_5\}$ *and the corresponding* basic *schema variants in Table 1.1. The user wants to get all employee names across all variants. They express this intent by the query* q_3*:*

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

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

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

In principle, variational queries can also express arbitrarily different intents

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

4.2 VRA Semantics

pls read the following paragraph

We use the semantics of relational queries to define the semantics of variational queries. ~~To this end~~ we first define the configuration function for variational queries which takes a configuration and a variational query and returns a relational query. We also define another version of the variational query configuration function that generates unique relational queries. Then, we define an accumulation function that accumulates multiple relational tables into a variational table.

complete this when you finish semantics.

4.2.1 VRA Configuration

The *configuration* function maps a variational query under a configuration to a relational query, defined in Figure 4.2. Thus, a variational query can be understood as a set of relational queries that their results is gathered in a single table and tagged with the feature expression stating their variant. Users can deploy queries for a specific variant by configuring them. Example 4.2.1 illustrates configuring a query and Example 4.2.2 illustrates the configuration of query q_1' from Example 4.1.1 and their corresponding relational results table.

added this example. pls review it.

Variational condition configuration:

$$
\begin{aligned}
\mathbb{C}[\![.]\!] &: \textbf{Condition} \rightarrow \textbf{Config} \rightarrow \underline{\textbf{Condition}} \\
\mathbb{C}[\![b]\!]_c &= b \\
\mathbb{C}[\![a \bullet k]\!]_c &= a \bullet k \\
\mathbb{C}[\![a_1 \bullet a_2]\!]_c &= a_1 \bullet a_2 \\
\mathbb{C}[\![\neg\theta]\!]_c &= \neg\mathbb{C}[\![\theta]\!]_c \\
\mathbb{C}[\![\theta_1 \vee \theta_2]\!]_c &= \mathbb{C}[\![\theta_1]\!]_c \vee \mathbb{C}[\![\theta_2]\!]_c \\
\mathbb{C}[\![\theta_1 \wedge \theta_2]\!]_c &= \mathbb{C}[\![\theta_1]\!]_c \wedge \mathbb{C}[\![\theta_2]\!]_c \\
\mathbb{C}[\![e\langle\theta_1, \theta_2\rangle]\!]_c &= \begin{cases} \mathbb{C}[\![\theta_1]\!]_c, & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \mathbb{C}[\![\theta_2]\!]_c, & \text{otherwise} \end{cases}
\end{aligned}
$$

Variational query configuration:

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

Figure 4.2: Configuration of variational queries and conditions. The configuration function assumes that the input, either the variational query or the variational condition, is well-typed.

Example 4.2.1. *Assume the underlying VDB has the variational schema* $S_3 = \{r\left(a_1{}^{f_1}, a_2, a_3\right)^{f_1 \vee f_2}\}$ *and the feature space* **FeatName** $= \{f_1, f_2\}$. *For valid configurations of this VDB* ($\{f_1, f_2\}$, $\{f_1\}$, $\{f_2\}$, *and* $\{\ \}$), *the variational query* $q_5 = \pi_{a_1, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2}}(r)$ *is configured to the following relational queries:* $\mathbb{Q}[\![q_5]\!]_{\{f_1\}} = \mathbb{Q}[\![q_5]\!]_{\{\ \}} = \pi_{a_1}\underline{r}$, $\mathbb{Q}[\![q_5]\!]_{\{f_2\}} = \pi_{a_1, a_3}\underline{r}$, $\mathbb{Q}[\![q_5]\!]_{\{f_1, f_2\}} = \pi_{a_1, a_2, a_3}\underline{r}$.

Example 4.2.2. *Consider the query* q_1' *given in Example 4.1.1:*

pls read this example.

$$q_1' = \pi_{empno^{(V_4 \vee V_5) \wedge \neg V_3}, name^{V_4 \wedge \neg V_3 \wedge \neg V_5}, firstname^{V_5 \wedge \neg V_3 \wedge \neg V_4}, lastname^{V_5 \wedge \neg V_3 \wedge \neg V_4}}(empbio).$$

Configuring q_1' *for all valid configurations* ($\{V_3\}$, $\{V_4\}$, $\{V_5\}$) *of the given VDB results in three relational queries:*

try to fit the result tables in the same page.

$$\mathbb{Q}[\![q_1']\!]_{\{V_3\}} = \varepsilon$$

$$\mathbb{Q}[\![q_1']\!]_{\{V_4\}} = \pi_{empno, name}(empbio)$$

$$\mathbb{Q}[\![q_1']\!]_{\{V_5\}} = \pi_{empno, firstname, lastname}(empbio)$$

Table 4.2 shows the result of these relational queries.

pls read the following paragraph, Figure 4.3, and Example 4.2.3 and Example 4.2.4.

Unfortunately, the configuration of variational queries may result in duplicate relational queries. In practice, this is not very efficient, as discussed later in Section 6.2. Thus, we define the *unique configuration* function $\mathcal{Q}(.)$: **Q** $\rightarrow$ **Set (Var Q)** that takes a variational query and returns a set of configured relational queries annotated with a presence condition. The presence condition is a feature expression generated from the set of configurations that configured the variational query into the same relational query. In essence, unique configuration can be defined for all data types that encode variation. For example, the unique

maybe call this "unique variants" instead?

Table 4.2: Results of relational queries from configuring the variational query q_1'.

(a) Result of the query $\mathbb{Q}[\![q_1']\!]_{\{V_3\}} = \varepsilon$.

result

(b) Result of the query $\mathbb{Q}[\![q_1']\!]_{\{V_4\}} = \pi_{empno,name}(empbio)$.

result

empno	*name*
80001	Nagui Merli
80002	Mayuko Meszaros
80003	Theirry Viele
...	...

(c) Result of the query $\mathbb{Q}[\![q_1']\!]_{\{V_5\}} = \pi_{empno,firstname,lastname}(empbio)$.

result

empno	*firstname*	*lastname*
200001	Selwyn	Koshiba
200002	Bedrich	Markovitch
200003	Pascal	Benzmuller
...	...	...

configuration function for variational queries can be defined by

$$\mathcal{Q}(q) = \{\underline{q}^e \mid \forall c \in \textbf{Config} : \mathbb{E}[\![e]\!]_c = \texttt{true}, \mathbb{Q}[\![q]\!]_c = \underline{q}\}.$$

Consequently, the unique configuration for variational sets of attributes (i.e., $\mathcal{A}(.)$: **Set (Var AttrName)→Var (Set AttrName)**) and variational conditions (i.e., $\mathcal{C}(.)$: **Condition→Var Condition**) are defined similarly. However, this definition is not efficient since it is still enumerating all possible configurations. Thus, we define the more efficient unique configuration function for variational queries in Figure 4.3.

see if you have more efficient $\mathcal{A}(A)$ and $\mathcal{C}(\theta)$

Example 4.2.3. *Consider the query* $q_5 = \pi_{a_1,a_2{}^{f_1 \wedge f_2},a_3{}^{f_2}}(r)$ *given in Example 4.2.1.*

$$
\begin{aligned}
\mathcal{Q}(.) &: \mathbf{Q} \rightarrow \mathbf{Set\ (Var\ \underline{Q})} \\
\mathcal{Q}(r) &= \{\underline{r}^{\mathtt{true}}\} \\
\mathcal{Q}(\sigma_\theta q) &= \{\left(\sigma_{\underline{\theta}}\underline{q}\right)^{e \wedge e_\theta} \mid \underline{q}^e \in \mathcal{Q}(q), \underline{\theta}^{e_\theta} \in \mathcal{C}(\theta)\} \\
\mathcal{Q}(\pi_A q) &= \{\left(\pi_{\underline{A}}\underline{q}\right)^{e \wedge e_A} \mid \underline{q}^e \in \mathcal{Q}(q), \underline{A}^{e_A} \in \mathcal{A}(A)\} \\
\mathcal{Q}(q_1 \times q_2) &= \{\left(\underline{q}_1 \times \underline{q}_2\right)^{e_1 \wedge e_2} \mid \underline{q}_1^{e_1} \in \mathcal{Q}(q_1), \underline{q}_2^{e_2} \in \mathcal{Q}(q_2)\} \\
\mathcal{Q}(q_1 \bowtie_\theta q_2) &= \{\left(\underline{q}_1 \bowtie_{\underline{\theta}} \underline{q}_2\right)^{e_1 \wedge e_2 \wedge e_\theta} \mid \underline{q}_1^{e_1} \in \mathcal{Q}(q_1), \underline{q}_2^{e_2} \in \mathcal{Q}(q_2), \underline{\theta}^{e_\theta} \in \mathcal{C}(\theta)\} \\
\mathcal{Q}(e\langle q_1, q_2\rangle) &= \{\underline{q}_1^{e \wedge e_1} \mid \underline{q}_1^{e_1} \in \mathcal{Q}(q_1)\} \cup \{\underline{q}_2^{\neg e \wedge e_2} \mid \underline{q}_2^{e_2} \in \mathcal{Q}(q_2)\} \\
\mathcal{Q}(q_1 \circ q_2) &= \{\left(\underline{q}_1 \circ \underline{q}_2\right)^{e_1 \wedge e_2} \mid \underline{q}_1^{e_1} \in \mathcal{Q}(q_1), \underline{q}_2^{e_2} \in \mathcal{Q}(q_2)\} \\
\mathcal{Q}(\varepsilon) &= \varepsilon^{\mathtt{true}}
\end{aligned}
$$

Figure 4.3: Unique configuration of variational queries. The unique configuration function assumes that the input is well-typed.

The unique configuration of this query results in the following set of queries:

$$\mathcal{Q}(q_5) = \{(\pi_{a_1}(\underline{r}))^{(f_1 \wedge \neg f_2) \vee (\neg f_1 \wedge \neg f_2)}, (\pi_{a_1,a_3}(\underline{r}))^{\neg f_1 \wedge f_2}, (\pi_{a_1,a_2,a_3}(\underline{r}))^{f_1 \wedge f_2}\}.$$

Example 4.2.4. *Consider the query q_1' configured in Example 4.2.2:*

$$q_1' = \pi_{empno^{(V_4 \vee V_5) \wedge \neg V_3}, name^{V_4 \wedge \neg V_3 \wedge \neg V_5}, firstname^{V_5 \wedge \neg V_3 \wedge \neg V_4}, lastname^{V_5 \wedge \neg V_3 \wedge \neg V_4}}(empbio).$$

The unique configuration of it results in:

$$\begin{aligned}
\mathcal{Q}(q_1') = \{&\varepsilon^{V_3 \wedge \neg V_4 \wedge \neg V_5}, (\pi_{empno,name}(empbio))^{\neg V_3 \wedge V_4 \wedge \neg V_5} \\
&, (\pi_{empno,firstname,lastname}(empbio))^{\neg V_3 \wedge V_4 \wedge \neg V_5}\}.
\end{aligned}$$

4.2.2 Accumulation of Relational Tables to a Variational Table

pls read this entire section. thx!

After connecting variational queries to relational queries, to define the semantics of VRA we need to connect the results of multiple relational queries to the result of a single variational query. Since we have two approaches to connect a variational query to relational queries we define two *accumulation* functions that generate a variational table from a set of relational tables. The first accumulation function *accum* : **Set FeatName** $\rightarrow$ **Set (Config,** $\underline{\textbf{Table}}$**)** $\rightarrow$ **Table** takes the feature space of a database and a set of relational tables with their attached configurations and generates a variational table. Figure 4.4 defines this function in terms of some auxiliary functions. The *mkTable* function takes a variational relation schema and a set of variational relation contents and generates a variational table that has the given schema and the variational tuples in the input tables. The *addPresCondToConfTables* function maps the *addPresCondToConfContent* over a set of tables and their attached configuration and the *addPresCondToConfContent* function adds the *prescond* attribute to a relational table and its corresponding value which is a feature expression associated with the given configuration using the closed set of features. The *fitConfTablesToVsch* maps the function *fitTableToVsch* to tables of a set of relational tables and their attached configuration. The *fitTableToVsch* function adjusts a table, both its schema and content, to a variational relation schema. The *tablesToVsch* maps the function *schToVsch* to a set of relational tables and their attached configuration. The *schToVsch* generates a variational relation schema from a set of plain relation schema and their

Table accumulation function:

$$
\begin{aligned}
&\mathit{accum} : \textbf{Set FeatName} \rightarrow \textbf{Set (Config, }\underline{\textbf{Table}}\textbf{)} \rightarrow \textbf{Table} \\
&\mathit{accum}\ \mathit{fs}\ \mathit{ts} = \mathit{mkTable}\ \mathit{vsch}\ \mathit{tables} \\
&\quad \mathit{where}\ \mathit{vsch} = \mathit{tablesToVsch}\ \mathit{fs}\ \mathit{ts} \\
&\qquad \mathit{tables} = \mathit{addPresCondToConfTables}\ \mathit{fs}\ \mathit{fitted} \\
&\qquad \mathit{fitted} = \mathit{fitConfTablesToVsch}\ \mathit{ts}\ \mathit{vsch}
\end{aligned}
$$

Auxiliary functions for table accumulation:

$$
\begin{aligned}
\mathit{schToVsch} &: \textbf{Set FeatName} \rightarrow \textbf{Set (Config, }\underline{\textbf{RelSch}}\textbf{)} \rightarrow \textbf{RelSch} \\
\mathit{tablesToVsch} &: \textbf{Set FeatName} \rightarrow \textbf{Set (Config, }\underline{\textbf{Table}}\textbf{)} \rightarrow \textbf{RelSch} \\
\mathit{fitTableToVsch} &: \underline{\textbf{Table}} \rightarrow \textbf{RelSch} \rightarrow \underline{\textbf{Table}} \\
\mathit{fitConfTablesToVsch} &: \textbf{Set (Config, }\underline{\textbf{Table}}\textbf{)} \rightarrow \textbf{RelSch} \rightarrow \textbf{Set (Config, }\underline{\textbf{Table}}\textbf{)} \\
\mathit{addPresCondToConfContent} &: \textbf{Set FeatName} \rightarrow \textbf{(Config, }\underline{\textbf{RelCont}}\textbf{)} \rightarrow \textbf{RelCont} \\
\mathit{addPresCondToConfTables} &: \textbf{Set FeatName} \rightarrow \textbf{Set (Config, }\underline{\textbf{Table}}\textbf{)} \rightarrow \textbf{Set RelCont} \\
\mathit{mkTable} &: \textbf{RelSch} \rightarrow \textbf{Set RelCont} \rightarrow \textbf{Table}
\end{aligned}
$$

Figure 4.4: Accumulation function of a set of relational tables with their attached configuration into a variational table and its auxiliary functions. The definition uses spaces to pass parameters. For example, $f\ x$ states that the parameter x is passed to the function x and $f\ x\ y$ states that parameters x and y are passed to f as the first and second arguments, respectively.

attached configuration given the closed set of features of the database's feature space.[1] Note that to generate a feature expression from a configuration it is essential to pass the closed set of features. Example 4.2.5 illustrates the behavior of these auxiliary functions and the table accumulation function over the relational tables in Table 4.2.

Example 4.2.5. *Consider the query q_1' written over the VDB with variational*

[1] In implementation, for efficiency, we pass the type of the query from VRA's type system as the variational relation schema that is generated by the *tablesToVsch* function.

schema S_2 *and feature space* $\mathbf{FeatName} = \{V_3, V_4, V_5\}$*, all given in Example 4.1.1. All configured relational queries of* q_1' *for VDDB's valid configurations and their corresponding results in form of a relational table are given in Example 4.2.2 and Table 4.2, respectively. Now we show how the relational tables of the configured queries, shown in Table 4.2, are accumulated to the variational table, shown in Table 4.1b, as the result of the variational query* q_1' *by using the table accumulation function accum. As the first step of accumulation, we generate the variational relation schema by applying tablesToVsch to tables in Table 4.2. This results in the variational relation schema* s_{accum}

$$s_{accum} = result(empno^{(\neg V_3 \wedge V_4 \wedge \neg V_5) \vee (\neg V_3 \wedge \neg V_4 \wedge V_5)}, name^{\neg V_3 \wedge V_4 \wedge \neg V_5}, \\ firstname^{\neg V_3 \wedge \neg V_4 \wedge V_5}, lastname^{\neg V_3 \wedge \neg V_4 \wedge V_5})^{oneof(V_3, V_4, V_5)}$$

Note that the presence conditions are generated based on the configurations attached to the tables. For example, the presence condition $(\neg V_3 \wedge V_4 \wedge \neg V_5) \vee (\neg V_3 \wedge \neg V_4 \wedge V_5)$ *associated with the attribute empno is the disjunction of two feature expressions* $(\neg V_3 \wedge V_4 \wedge \neg V_5)$ *and* $(\neg V_3 \wedge \neg V_4 \wedge V_5)$ *where they represent the configuration* $\{V_4\}$ *(associated to Table 4.2b) and* $\{V_5\}$ *(associated to Table 4.2c), respectively. That is, the configuration* $\{\ \}V_4$ *represents the variants that only enable the feature* V_4 *from* V_3*–*V_5*, thus, its corresponding feature expression is* $(\neg V_3 \wedge V_4 \wedge \neg V_5)$*. That is why we need to pass the closed set of features to the auxiliary functions (to generate feature expression corresponding to configurations).*

In the next step, the tables in Table 4.2 are adjusted so that they all match a

certain relation schema. This is achieved by the fitConfTablesToVsch which gets all the tables in Table 4.2 with their associated configurations and the variational relation schema generated by passing them to the tablesToVsch. This is done by mapping the fitTableToVsch to all the tables in Table 4.2 with their associated configurations. This function simply adds attributes of the variational relation schema to the table that do not exists in the table and puts `NULL` *as values in the tuples for those attributes. Table 4.3a–Table 4.3c illustrate the application of fitTableToVsch to Table 4.2a–Table 4.2c and variational relation schema* s_{accum}.

Table 4.3: Step two of table accumulation applies the *fitConfTablesToVsch* function to all tables of Table 4.2 and their corresponding configurations and the variational relation schema s_{accum}.

(a) Result of the *fitTableToVsch* applied to Table 4.2a and variational relation schema s_{accum}.

result	*empno*	*name*	*firstname*	*lastname*

(b) Result of the *fitTableToVsch* applied to Table 4.2b and variational relation schema s_{accum}.

result	*empno*	*name*	*firstname*	*lastname*
	80001	Nagui Merli		
	80002	Mayuko Meszaros		
	80003	Theirry Viele		
	...	...	...	...

(c) Result of the *fitTableToVsch* applied to Table 4.2c and variational relation schema s_{accum}.

result	*empno*	*name*	*firstname*	*lastname*
	200001		Selwyn	Koshiba
	200002		Bedrich	Markovitch
	200003		Pascal	Benzmuller
	...	...	...	...

Then, the addPresCondToConfContent function adds the presence condition attribute and its values to relation contents of tables in Table 4.3, resulting in Table 4.4 which illustrates a set of relation contents that are separated by the red bold line.

Table 4.4: Step three of table accumulation adds the presence condition values to relation contents. The table illustrates a set of relation contents that are separated by the red bold line between them. The tuples follow the order of attributes in the relation schema.

					$V_3 \wedge \neg V_4 \wedge \neg V_5$
	80001	Nagui Merli			$\neg V_3 \wedge V_4 \wedge \neg V_5$
	80002	Mayuko Meszaros			$\neg V_3 \wedge V_4 \wedge \neg V_5$
	80003	Theirry Viele			$\neg V_3 \wedge V_4 \wedge \neg V_5$
	...	...	...	...	...
	200001		Selwyn	Koshiba	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	200002		Bedrich	Markovitch	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	200003		Pascal	Benzmuller	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	...	...	...	...	...

Finally, the mkTable function takes the variational relation schema s_{accum} and Table 4.4. Note that the values in tuples of Table 4.4 follow the order of the attributes in the variational relation schema. This results in Table 4.5 which is equivalent to the result of q_1' given in Table 4.1b.

The second accumulation function $accum'$: **Set (Var Table) → Table** takes a set of relational tables that are annotated with a feature expression instead of their attached configuration. Figure 4.5 defines this function and its auxiliary functions. The auxiliary functions are similar to the ones defined in Figure 4.5 except that they do not need to generate a feature expression from a configuration

Table 4.5: Final step of table accumulation passes the variational relation schema s_{accum} and relation contents in Table 4.4 to the *mkTable* function.

$oneof\ (V_3, V_4, V_5)$	$(\neg V_3 \wedge V_4 \wedge \neg V_5)$				
result	$(\vee(\neg V_3 \wedge \neg V_4 \wedge V_5)$	$\neg V_3 \wedge V_4 \wedge \neg V_5$	$\neg V_3 \wedge \neg V_4 \wedge V_5$	$\neg V_3 \wedge \neg V_4 \wedge V_5$	`true`
	empno	*name*	*firstname*	*lastname*	*prescond*
	80001	Nagui Merli			$V_3 \wedge \neg V_4 \wedge \neg V_5$
	80002	Mayuko Meszaros			$\neg V_3 \wedge V_4 \wedge \neg V_5$
	80003	Theirry Viele			$\neg V_3 \wedge V_4 \wedge \neg V_5$
	200001		Selwyn	Koshiba	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	200002		Bedrich	Markovitch	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	200003		Pascal	Benzmuller	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	...	...	...	...	...

and a set of closed features.

4.2.3 VRA Denotational Semantics

pls read this entire section. thx!

Now that we have all required parts we define the denotational semantics of variational relational algebra using the denotational semantics of relational algebra. The denotational semantics of relational queries $rqSem : \underline{\mathbf{Q}} \rightarrow \underline{\mathbf{DBInst}} \rightarrow \underline{\mathbf{Table}}$ takes a plain query and a plain database and returns a table named *result*. We do not give a formal definition of *rqSem*, however, examples of the semantics of a query are given in Table 4.2. We then define the VRA denotational semantics $vqSem$: $\mathbf{Q} \rightarrow \mathbf{DBInst} \rightarrow \mathbf{Table}$ as the accumulation of relational tables resulted from the semantics of its configured queries over their corresponding configured databases for all valid configurations of a variational database. The *mapRQSem* takes a set of relational queries with their attached configurations and a variational database instance and returns the set of query semantics over their configured database with their attached configurations, that is, it maps *rqSem* on the relational queries over their corresponding relational database.[2]

[2]In implementation, the closed set of features and valid configurations of a VDB are contained

Table accumulation function:

$$
\begin{aligned}
&accum' : \textbf{Set (Var } \underline{\textbf{Table}}\textbf{)} \rightarrow \textbf{Table} \\
&accum'\ fs\ ts = mkTable\ vsch\ tables \\
&\quad where\ vsch = annotTablesToVsch\ ts \\
&\qquad tables = addPresCondToVarTables\ fitted \\
&\qquad fitted = fitVarTablesToVsch\ ts\ vsch
\end{aligned}
$$

Auxiliary functions for table accumulation:

$$
\begin{aligned}
annotSchToVsch &: \textbf{Set (Var } \underline{\textbf{RelSch}}\textbf{)} \rightarrow \textbf{RelSch} \\
annotTablesToVsch &: \textbf{Set (Var } \underline{\textbf{Table}}\textbf{)} \rightarrow \textbf{RelSch} \\
fitVarTablesToVsch &: \textbf{Set (Var } \underline{\textbf{Table}}\textbf{)} \rightarrow \textbf{RelSch} \rightarrow \textbf{Set (Var } \underline{\textbf{Table}}\textbf{)} \\
addPresCondToVarContent &: \textbf{Var } \underline{\textbf{RelCont}} \rightarrow \textbf{RelCont} \\
addPresCondToVarTables &: \textbf{Set (Var } \underline{\textbf{Table}}\textbf{)} \rightarrow \textbf{Set RelCont}
\end{aligned}
$$

Figure 4.5: Accumulation function of a set of relational tables annotated with a feature expression into a variational table and its auxiliary functions. The definition uses spaces to pass parameters, e.g., $f\ x = f(x)$ and $f\ x\ y = f(x, y)$.

VRA denotational semantics:

$$
\begin{aligned}
&\mathit{vqSem} : \mathbf{Q} \rightarrow \mathbf{DBInst} \rightarrow \mathbf{Table} \\
&\mathit{vqSem}\ q\ \mathcal{I} = \mathit{accum}\ \mathit{fs}\ \mathit{tabs} \\
&\quad \mathit{where}\ \mathit{fs} = \mathit{featues}\ \mathcal{I} \\
&\qquad\quad \mathit{rqs} = \mathit{qToConfRelQs}\ q\ (\mathit{validConfigs}\ \mathcal{I}) \\
&\qquad\quad \mathit{tabs} = \mathit{mapRQSem}\ \mathit{rqs}\ \mathcal{I}
\end{aligned}
$$

Auxiliary functions for VRA denotational semantics:

$$
\begin{aligned}
\mathit{rqSem} &: \underline{\mathbf{Q}} \rightarrow \underline{\mathbf{DBInst}} \rightarrow \underline{\mathbf{Table}} \\
\mathit{mapRQSem} &: \mathbf{Set\ (Config, \underline{Q})} \rightarrow \mathbf{DBInst} \rightarrow \mathbf{Set\ (Config, \underline{Table})} \\
\mathit{features} &: \mathbf{DBInst} \rightarrow \mathbf{Set\ FeatName} \\
\mathit{validConfigs} &: \mathbf{DBInst} \rightarrow \mathbf{Set\ Config} \\
\mathit{qToConfRelQs} &: \mathbf{Q} \rightarrow \mathbf{Set\ Config} \rightarrow \mathbf{Set\ (Config, \underline{Q})}
\end{aligned}
$$

Figure 4.6: Denotational semantics of variational relational algebra.

4.3 VRA Type System

[type sys] [work on the derivation tree example separate from the text since youre waiting for eric on the text]

4.4 Explicitly Annotating Queries

Variational queries do not need to repeat information that can be inferred from the variational schema or the type of a query. For example, the query q_1 shown in Example 4.1.1 does not contradict the schema and thus is type correct. However, it does not include the presence conditions of attributes and the relation encoded in the schema while q_6 repeats this information:

$q_6 = \pi_{empno^{(V_4 \vee V_5) \wedge \neg V_3}, name^{\neg V_3 \wedge V_4 \wedge \neg V_5}, firstname^{\neg V_3 \wedge \neg V_4 \wedge V_5}, lastname^{\neg V_3 \wedge \neg V_4 \wedge V_5}} (e_2\langle empbio, \varepsilon \rangle)$.[3]

I added the footnote to be more explicit about the example.

Similarly, the projection in the query $q_7 = \pi_{name, firstname}(subq_7)$ where $subq_7 = V_4\langle \pi_{name}(q_6), \pi_{firstname}(q_6) \rangle$ is written over S_2 and it does not repeat the presence conditions of attributes from its $subq_7$'s type. The query $q_8 = \pi_{name^{V_4}, firstname^{\neg V_4}}(subq_7)$ makes the annotations of projected attributes *explicit* with respect to both the variational schema S_2 and its subquery's type. Although relieving the user from explicitly repeating variation makes VRA easier to use, queries still have to state variation explicitly to avoid losing information when decoupled from the schema.

within, instead of extracting them from the database. However, we keep the formalization simple and assume that they can also be retrieved from the VDB.

[3]The query q_6 is the simplified version of $\lfloor q_1 \rfloor_{S_2} = \pi_{empno^{(V_4 \vee V_5) \wedge \neg V_3}, name^{\neg V_3 \wedge V_4 \wedge \neg V_5}, firstname^{\neg V_3 \wedge \neg V_4 \wedge V_5}, lastname^{\neg V_3 \wedge \neg V_4 \wedge V_5}} (\lfloor empbio \rfloor_{S_2})$ where $\lfloor empbio \rfloor_{S_2} = e_2\langle \pi_{empno, name^{V_4}, firstname^{V_5}, lastname^{V_5}}(empbio), \varepsilon \rangle$.

Variational queries typing rules:

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

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

$$\textsc{Select-E}\quad \frac{e, S \vdash q : A^{e'} \qquad e, \downarrow\left(A^{e'}\right) \vdash \theta}{e, S \vdash \sigma_\theta q : A^{e'}} \qquad \textsc{Choice-E}\quad \frac{e \wedge e', S \vdash q_1 : {A_1}^{e_1} \qquad e \wedge \neg e', S \vdash q_2 : {A_2}^{e_2}}{e, S \vdash e'\langle q_1, q_2\rangle : (\downarrow(A_1^{e_1}) \cup \downarrow(A_2^{e_2}))^{(e_1)\vee(e_2)}}$$

$$\textsc{Product-E}\quad \frac{e, S \vdash q_1 : {A_1}^{e_1} \qquad e, S \vdash q_2 : {A_2}^{e_2} \qquad \downarrow(A_1^{e_1}) \cap \downarrow(A_2^{e_2}) = \{\}}{e, S \vdash q_1 \times q_2 : (\downarrow(A_1^{e_1}) \cup \downarrow(A_2^{e_2}))^{e_1 \wedge e_2}}$$

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

Variational condition typing rules:

$$\textsc{Boolean-C}\quad e, A \vdash b \qquad \textsc{Conjunction-C}\quad \frac{e, A \vdash \theta_1 \qquad e, A \vdash \theta_2}{e, A \vdash \theta_1 \wedge \theta_2} \qquad \textsc{Disjunction-C}\quad \frac{e, A \vdash \theta_1 \qquad e, A \vdash \theta_2}{e, A \vdash \theta_1 \vee \theta_2}$$

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

$$\textsc{AttOptVal-C}\quad \frac{a^{e'} \in A \qquad sat\left(e' \wedge e\right)}{e, A \vdash a \bullet k} \qquad \textsc{AttOptAtt-C}\quad \frac{{a_1}^{e_1} \in A \qquad {a_2}^{e_2} \in A \qquad sat(e_1 \wedge e_2 \wedge e)}{e, A \vdash a_1 \bullet a_2}$$

Figure 4.7: VRA and variational condition typing relation. The rules assume that the underlying VDB is well-formed. Remember that our theory assumes all attributes have the same type and all constants belong to attributes' domain.

Figure 4.8: Derivation tree for q_1 in Example **??**.

Assumption 1:

$$\frac{empbio(empno, sex, birthdate, name^{V_4}, firstname^{V_5}, lastname^{V_5}) \in S_2 \qquad sat(e_2 \wedge e_2)}{e_2, S_2 \vdash empbio : (empno, sex, birthdate, name^{V_4}, firstname^{V_5}, lastname^{V_5})^{e_2}} \text{ RELATION-E}$$

Assumption 2:

$$\{empno^{V_4 \vee V_5}, name, firstname, lastname\} \\ \prec\!\downarrow\Big(\{empno, sex, birthdate, name^{V_4}, firstname^{V_5}, lastname^{V_5}\}^{e_2}\Big)$$

Assumption 3:

$$|\downarrow\Big(\{empno^{V_4 \vee V_5}, name, firstname, lastname\}^{e_2}\Big)| = |\{empno^{V_4 \vee V_5}, name, firstname, lastname\}|$$

Final derivation tree:

$$\frac{\textbf{Assumption 1} \qquad \textbf{Assumption 2} \qquad \textbf{Assumption 3}}{e_2, S_2 \vdash q_1 : (empno^{(V_4 \vee V_5)}, name^{V_4}, firstname^{V_5}, lastname^{V_5})^{e_2}} \text{ PROJECT-E}$$

We do this by defining the function $\lfloor q \rfloor_S : \mathbf{Q} \rightarrow \mathbf{Sch} \rightarrow \mathbf{Q}$, that *explicitly annotates a query q with the schema S*. The explicitly annotating query function, formally defined in Figure 4.10, conjoins attributes and relations presence conditions with the corresponding annotations in the query and wraps subqueries in a choice when needed. Note that, q_8 and q_6 are the result of $\lfloor q_7 \rfloor_{S_2}$ and $\lfloor q_1 \rfloor_{S_2}$, respectively, after simplification [4].

Theorem 4.4.1. *If the query q has the type A then its explicitly annotated counterpart has the same type A, i.e.:*

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

This shows that the type system applies the schema to the type of a query although it

[4] More specifically, they are simpilified using rules defined in Figure 4.11

Figure 4.9: Derivation tree for q_2 in Example **??**.

Assumption 1:

$$empbio(empno, sex, birthdate, name^{V_4}, firstname^{V_5}, lastname^{V_5})^{e_2} \in S_2$$

Assumption 2:

$$\frac{\textbf{Assumption 1} \qquad sat((e_2 \wedge (\neg V_3)) \wedge e_2)}{e_2 \wedge (\neg V_3), S_2 \vdash empbio : (empno, sex, birthdate, name^{V_4}, firstname^{V_5}, lastname^{V_5})^{e_2 \wedge (\neg V_3)}} \textsc{Relation-E}$$

Assumption 3:

$$|\downarrow\Big(\{empno, name, firstname, lastname\}^{e_2 \wedge \neg V_3}\Big)| = |\{empno, name, firstname, lastname\}|$$

Assumption 4:

$$\{empno, name, firstname, lastname\} \prec \{empno^{e_2 \wedge (\neg V_3)}, sex^{e_2 \wedge (\neg V_3)}, birthdate^{e_2 \wedge (\neg V_3)}, name^{V_4}, firstname^{V_5}, lastname^{V_5}\}$$

Assumption 5 (derivation tree for $left = \pi_{empno,name,firstname,lastname}(empbio)$):

$$\frac{\textbf{Assumption 2} \qquad \textbf{Assumption 3} \qquad \textbf{Assumption 4}}{e_2 \wedge (\neg V_3), S_2 \vdash left : (empno, name^{V_4}, firstname^{V_5}, lastname^{V_4})^{e_2 \wedge (\neg V_3)}} \textsc{Project-E}$$

Final derivation tree for q_2:

$$\frac{\textbf{Assumption 5} \qquad \dfrac{}{e_2 \wedge \neg(\neg V_3), S_2 \vdash \varepsilon : \{\ \}^{\texttt{false}}} \textsc{EmptyRelation-E}}{e_2, S_2 \vdash q_2 : (empno, name^{V_4}, firstname^{V_5}, lastname^{V_5})^{(e_2 \wedge (\neg V_3)) \vee \texttt{false}}} \textsc{Choice-E}$$

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

Figure 4.10: Explicitly annotating a well-typed query with a variational schema.

does not apply it to the query. The type equivalence *is variational set equivalence, defined in Figure 2.4, for normalized variational sets of attributes.*

We encode and prove Theorem 4.4.1 in the Coq proof assistant [**?**]. We also illustrate the application of Theorem 4.4.1 to queries q_1 and q_6. Example **??** explained how q_1's type is generated step-by-step. The variation context and underlying schema are the same and the subquery *empbio* has the same type. The projected attribute set annotated with the variation context is:
$A_2 = \{empno^{(V_4 \vee V_5) \wedge \neg V_3}, name^{\neg V_3 \wedge V_4 \wedge \neg V_5}, firstname^{\neg V_3 \wedge \neg V_4 \wedge V_5}, lastname^{\neg V_3 \wedge \neg V_4 \wedge V_5}\}^{e_2}$,
which is clearly subsumed by A_{empbio}, thus, its intersection with A_{empbio} annotated with the presence condition of A_{empbio} is itself, hence, $A_{q_1} \equiv A_{q_6}$.

added this paragraph. pls review it

Explicitly annotating variational queries not only relieves the user from repeating the database's variation in their queries but it is also necessary for the functions that take a query without taking the schema such as the query configuration function. This is as opposed to other functions that have to take both the query and

the schema such as the type system. Example 4.4.2 illustrates why a query passed to the configuration function must be explicitly annotated.

Example 4.4.2. *Consider the variational query* $q_5 = \pi_{a_1,a_2{}^{f_1 \wedge f_2},a_3{}^{f_2}}(r)$ *given in Example 4.2.1. This query is not explicitly annotated since attribute* a_1 *does not carry its variational encoding from the database, that is, it does not have the presence condition* f_1*. Explicitly annotating this query gives us query* $q_5' = \pi_{a_1{}^{f_1},a_2{}^{f_1 \wedge f_2},a_3{}^{f_2}}(r)$*. Configuring* q_5' *results in the same query as configuring* q_5 *except for configuration* $\{\ \}$*, that is,* $\mathbb{Q}[\![q_5']\!]_{\{\ \}} = \pi_{\{\ \}}\underline{r} = \varepsilon$*. The reason why* $\mathbb{Q}[\![q_5]\!]_{\{\ \}}$ *is incorrect is that* q_5 *is missing the variation attached to attribute* a_1 *and the configuration function does not consider the schema of a database while configuring variational queries written over that database.*

4.5 Variation-Minimization Rules

add q_6 is simplified of q_6' because of rule application blah blah.

add example + more rules + point out interesting ones

VRA is flexible since an information need can be represented via multiple v-queries as demonstrated in Example 4.1.1 and Example 4.1.2. It allows users to incorporate their personal taste and task requirements into v-queries they write by having different levels of variation. For example, consider the explicitly annotated query q_6 in Section 4.4:
$q_6 = \pi_{empno^{V_4 \vee V_5},name^{V_4},firstname^{V_5},lastname^{V_5}}\ (m_2\langle empbio, \varepsilon\rangle)$. To be explicit about the exact query that will be run for each variant the query q_6's variation can be *lifted up* by using choices, resulting in the query q_6''
$q_6'' = V_4\langle \pi_{empno,name}\, empbio, V_5\langle \pi_{empno,firstname,lastname}\, empbio, \varepsilon\rangle\rangle$. While q_6 contains

Choice Distributive Rules:

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

CC and RA Optimization Rules:

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

Figure 4.11: Selected variation minimization rules.

less redundancy q_6'' is more comprehensible since the variants are explicitly stated in the dimension of the choice. Thus, *supporting multiple levels of variation creates a tension between reducing redundancy and maintaining comprehensibility.*

We define *variation minimization* rules in Figure 4.11 that are syntactic and preserve the semantics. Pushing in variation into a query, i.e., applying rules left-to-right, reduces redundancy while lifting them up, i.e., applying rules right-to-left, makes a query more understandable. When applied left-to-right, the rules are terminating since the scope of variation monotonically decreases in size.

4.6 Variational Relational Algebra Properties

pls read the following paragraph

In this section, we discuss important properties of VRA. We first discuss its

expressiveness with regards to the relational algebra in Section 4.6.1. Then, we discuss VRA's type safety in Section 4.6.2 by taking advantage of the relational algebra's type safety and defining a property that connect VRA's type system to relational algebra's type system, called *variation-preserving property*.

4.6.1 Expressiveness

VRA enables querying multiple database variants encoded as a singled VDB simultaneously and selectively. More precisely, VRA is *maximally expressive* in the sense that it can express any set of plain RA queries over any subset of relational database variants encoded as a VDB. We prove this claim in Theorem 4.6.1.

Theorem 4.6.1. *Given a set of plain RA queries* $\underline{q}_1, \ldots, \underline{q}_n$ *where each query* $\underline{q}_i$ *is to be executed over a disjoint subset* $\mathcal{I}_i$ *of variants of the VDB instance* $\mathcal{I}$*, there exists a variational query* q *such that* $\forall c \in \mathbf{Config}.\ \mathbb{I}[\![\mathcal{I}]\!]_c = \mathcal{I}_i \implies \mathbb{Q}[\![q]\!]_c = \underline{q}_i$.

Proof. By construction. Let f_i be the feature expression that uniquely characterizes the variants in each $\mathcal{I}_i$. Then

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

□

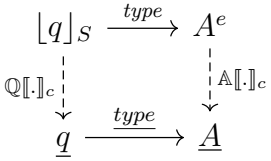
The above construction relies on the fact that every RA query is a valid VRA (sub)query in which every presence condition is `true`. Of course, in most realistic scenarios, we expect that variational queries can be encoded more efficiently by

sharing commonalities and embedding relevant choices and presence conditions within the variational query.

4.6.2 Type Safety

To show that VRA is type safe we benefit from RA's type safety [44] by defining the *variation-preserving* property for VRA which connects VRA to RA. The *variation-preserving property with respect to variational schema* states that if a query q has type A then configuring the type of a valid explicitly annotated query is the same as the type of its configured corresponding query. Theorem 4.6.2 proves this property.

Theorem 4.6.2 is visualized in the diagram below, where the vertical arrows indicate corresponding configure functions, *type* indicates VRA's type system, i.e., $type(q) = A^e$ is $S \vdash q : A^e$, and $\underline{type}$ indicates RA's type system, i.e., $\underline{S} \vdash \underline{q} : \underline{A}$. We assume that corresponding variation schema and schema is passed to type systems. Simply put, the relational type of the configured variational query q with configuration c, i.e., $\mathbb{A}[\![type(q)]\!]_c$, must be the same as the configured variational type of the variational query q with configuration c, i.e., $\underline{type}(\mathbb{Q}[\![q]\!]_c)$. *Clearly the diagram commutes*: taking either path of 1) configuring $\lfloor q \rfloor_S$ first and then obtaining the relational type of it or 2) obtaining the variational type of $\lfloor q \rfloor_S$ first and then configuring it results in the same set of attributes. The variation-preserving property enforces the maintenance of variants that a tuple belongs to through running a query at



the schema level.[5]. Example 4.6.3 illustrates why the query must be constrained by the variation schema in the variation-preserving diagram.

Theorem 4.6.2. *For all configurations c, if a query q has type A then its configured query* $\mathbb{Q}[\![\lfloor q \rfloor_S]\!]_c$ *has type* $\mathbb{A}[\![A]\!]_c$*, i.e.,*

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

Proof. By structural induction. We proved this theorem in the Coq proof assistant [**?**]. □

Example 4.6.3. *Consider the variational query* $q_5 = \pi_{a_1, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2}} r$ *given in Example 4.2.1. It is well-typed and it has the type* $A = \{a_1{}^{f_1}, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2}\}$*. Configuring A for the variant that both* f_1 *and* f_2 *are disabled results is an empty attribute set. However, the type of its configured query for this variant, i.e.,* $\mathbb{Q}[\![q_5]\!]_{\{\ \}} = \pi_{a_1} \underline{r}$*, is the attribute set* $\{a_1\}$*. This violates the variation-preserving property. A similar problem happens for the variant of* $\{f_2\}$*, i.e.,* $\underline{type}\left(\mathbb{Q}[\![q_5]\!]_{\{f_2\}}\right) = \underline{type}\left(\pi_{a_1, a_3} \underline{r}\right) = \{a_1, a_3\} \neq \{a_3\} = \mathbb{A}[\![A]\!]_{\{f_2\}} = \mathbb{A}[\![type\,(q_5)]\!]_{\{f_2\}}$*. However, the variation-preserving property holds for the constrained query by variation schema, i.e.,* $\lfloor q_5 \rfloor_{S_3} = \pi_{a_1{}^{f_1}, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2}} r$*. Thus, the input query to the configuration function* $\mathbb{Q}[\![.]\!]_c$ must *be explicitly annotated by the underlying variation schema for the configured query to match the underlying configured schema.*

do i need to add theorem for preserve + progress?

[5]We define this property as a test at the semantics level and show that all our experimental queries passed it.

Chapter 5 Variational Database Use Cases

pls read from here

Thus far we introduced the variational database framework and variational queries. However, the question becomes: "How useable is the variational database framework?", "How hard is it to generate a VDB and write variational queries?", "Can a VDB be generated automatically? And if so, what is required to make this process automated?". In this chapter, we aim at answering these questions. Thus, the goal of this chapter is twofold: first, how generating a VDB can be made automatic; second, to guid an expert through both generating a VDB from a variation scenario when it cannot be done automatically and writing variational queries that express expert's information need over multiple database variants in the variation scenario.

A VDB can be generated automatically when the main variational information and database variants are available, that is, when the closed set of features, the feature model, and closed set of database variants are provided. Unfortunately, in practice, these information and encodings are not available since existing work only focuses on studying a specific kind of variation in databases and does not encode variation inside the database, instead, it addresses the problem with tools that simulate the effect of the specific kind of variation. Consequently, 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

to here.

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 [52].

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, yet we provide the full query sets online.[1] 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

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

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.

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. 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 thesis for its brevity because it is much more concise.

also, pls read from here

We first focus on variation in databases over space, Section 5.1. Section 5.1.1 describes the variation scenario from Hall [30] that is the basis of this use case including the feature set and feature model. Then, Section 5.1.2 and Section 5.1.3 describe generating the variational schema for the described variation scenario and populating the email SPL VDB with Enron email data,[2] respectively. Finally, Section 5.1.4 describes how variational queries capture the information need adapted from feature interactions described by Hall [30]. We then switch focus to variation in databases over time, Section 5.2. Again, Section 5.2.1 describes the evolution

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

of an employee database as the variation scenario from Moon et al. [43] that is the basis of this use case. Then, Section 5.2.2 and Section 5.2.3 describe generating the variational schema for the described variation scenario and populating the employee VDB with a well-known employee data set,[3] respectively. Finally, Section 5.2.4 describes the adapted and adjusted queries from Moon et al. [43]. At the end of this chapter, Section 5.3, we discuss the trade offs of using variational databases and we answer the question: "Should variation be encoded explicitly in databases?". to here.

We distribute the VDBs, SQL scripts for generating them, and queries of our use cases.[4] 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

[3] `https://github.com/datacharmer/test_db`
[4] Available at: `https://zenodo.org/record/4321921`

code belongs to which feature(s) and supporting the process of configuring a SPL 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 [30]. 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 [49]. 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 [30] as high-level specification of a SPL.

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

[5] `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 [30]:

- *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 variational 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 $e_{en} = \texttt{true}$, given as a feature expression. The feature model e_{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 Variational Schema of the Email SPL VDB

To produce a variational 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 [30]. 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

Table 5.2: Variational schema of the email VDB with feature model e_{en}. Presence conditions are colored blue for clarity.

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

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

From this starting point, we introduce new attributes and relations that are needed to implement the features in the email SPL. We attach presence conditions to new attributes and relations corresponding to the features they are needed to

support, which ensure they will *not* be present in configurations that do not include the relevant features. The resulting variational 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 variational 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 [30] 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 [30], such as the *referenceinfo* relation and alternative email addresses.

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.Second, we provide the variational 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 relevant presence conditions: that of the variational schema itself (i.e. the feature model), and

those of each relation and attribute.[6] The *element_id* of the feature model is *variational_schema*; the *element_id* of a relation r is its name r, and of attribute a in relation r is $r.a$. The plain SQL encoding of the variational 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 variational schema [49]. 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 en-

[6] All encodings are available at: `https://zenodo.org/record/4321921`.

vironment 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.

We provide SQL scripts to automate the creation of views for each variantand to automate the population of these views with tuples from the original dataset,which also sets each tuple's presence condition. The resulting database is distributed in two forms, one with the embedded variational schema as described in Sec-

tion 5.1.2,and one without the embedded schemafor use with our VDBMS tool in which the variational schema is provided separately.[7] We have tested the email SPL VDB for the properties described in Section 3.3 and all of them hold.

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 [30].. 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

[7]Both the scripts and different encodings of the email SPL VDB are available at: `https://zenodo.org/record/4321921`.

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 $Q_{bf} = filtermessages\langle Q_{filter}, Q_{basic}\rangle$. Although we do not show the process in this thesis, we can use equivalence laws from the choice calculus [23, 34] 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.

$$Q_{forward} = \pi_{rvalue,forwardaddr,subject,body}\, temp$$

$$temp \leftarrow mes\_rec\_emp \bowtie_{employeelist.eid=forward\_msg.eid} auto\_msg$$

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

Besides single-feature queries, we also provide queries that gather information needed to identify and address the undesirable feature interactions described by Hall [30]. 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 two features *signature* and *forwardmessages*: 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.

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

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}.

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

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

5.2 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 [52], and dealing with such changes is a well-studied problem in the database community known as *database evolution* [46].

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. [43] and populated by a dataset that is widely used in databases research.[8]

5.2.1 Variation Scenario: An Evolving Employee Database

Moon et al. [43] 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 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

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

Table 5.3: Evolution of an employee database schema from Moon et al. [43].

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*)

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 e_{emp}. Also, note that the feature model represent a restriction on the entire database.

$$e_{emp} = oneof\ (V_1, V_2, V_3, V_4, V_5)$$

Table 5.4: Employee variational schema with feature model. e_{emp}.

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

5.2.2 Generating Variational Schema of the Employee VDB

The variational 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 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 e_{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*.

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.Second, we provide a direct encoding in SQL that generates the universal schema for the VDB in either MySQL or Postgres.The variability of the schema is embedded within the employee VDBusing the same encoding as described at the end of Section 5.1.2.[9]

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

[9]All encodings of the employee variational schema are available at: `https://zenodo.org/record/4321921`.

to the new schema [46]. 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 variational schema, yet it includes all required information. Thus, we manually mapped data from the original schema onto each version of the variational schema.

We provide SQL scripts of required queries to automatically generate the employee VDB. We also provide SQL scripts to automate the separation of each group of employees into views according to their hire dateand populating those views from data in the employee database.[10]

As for any VDB, if an attribute is not present in any of the variants covered by a tuple's presence condition, that attribute will be set to NULL in the tuple. We do this even though the relevant information may be contained in the original employee database to ensure that we have a consistent VDB. For example, while inserting tuples into the V_4 view of the *empbio* table, we always insert NULL values attributes *firstname* and *lastname*. 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.[11] We have tested the employee VDB for the properties described in Section 3.3 and all of them hold.

[10] All the scripts are available at: `https://zenodo.org/record/4321921`.
[11] Both formats are availabe at: `https://zenodo.org/record/4321921`.

5.2.4 Employee Query Set

For this use case, we have a set of existing plain queries to start from. Moon et al. [43] 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. We provide the queries in both the VRA format usable by VDBMS and as `#ifdef`-annotated SQL.[12] We give an example of query written as `#ifdef`-annotated at the end of this section. 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:

$$empQ_1 = \pi_{salary^{V_3}} (\sigma_{empno=10004} empacct) \bowtie_{empacct.title=job.title} job.$$

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., $({a_1}^{e_1}, {a_2}^{e_2})^e$.

[12] All queries are available at: `https://zenodo.org/record/4321921`.

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

$$
\begin{aligned}
empQ_2 = V_3 \vee V_4 \vee V_5\langle & \\
& \pi_{salary}(V_3 \vee V_4\langle((\sigma_{empno=10004}empacct)) \bowtie job, \sigma_{empno=10004}empacct\rangle) \\
& ,\varepsilon\rangle
\end{aligned}
$$

There are a variety of ways we could encod both $empQ_1$ and $empQ_2$. For $empQ_1$ 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 $empQ_2$ 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.

$$
\begin{aligned}
empQ'_2 = \pi_{salary^{V_3 \vee V_4 \vee V_5}} & \\
& (V_3 \vee V_4 \langle (\sigma_{empno=10004}\, empacct) \bowtie_{empacct.title=job.title} job \\
& \quad , V_5 \langle \sigma_{empno=10004}\, empacct, \varepsilon \rangle \rangle)
\end{aligned}
$$

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:

$$
\begin{aligned}
empQ_3 = V_3 \vee V_4 \vee V_5 \langle & \\
& \pi_{name,firstname,lastname}(V_3 \langle empacct, empbio \rangle \bowtie_{empno=managerno} \\
& \qquad (\sigma_{deptno=\text{“}d001\text{”}}\, dept)) \\
& , \varepsilon \rangle
\end{aligned}
$$

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, $empQ_3$ is equivalent to the following query in which the presence conditions of the attributes from the variational schema are listed explicitly in the

projection:

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

Allowing developers to encode variation in variational queries based on their preference makes VRA more flexible and easy to use. Also, variational 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 $empQ_3$ 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 * FROM dept WHERE deptno="d001")
  ON empno=manageno
```

5.3 Discussion: Should Variation Be Encoded Explicitly in Databases?

In this section we discuss the use cases and our encodings of VDB and variational 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 Chapter 5 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 variational 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 variational query.[13]

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

We expect that ill-supported forms of variation are common in industry and

[13]The expressiveness of VDBs and variational 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 variational queries, all variants can be organized under a tree of choices that similarly organizes the variants in the appropriate way.

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 variational 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 annota-

tions in annotative approaches to SPL implementation [37]. There is widespread acknowledgment that unrestricted use of variation annotations, such as the C Preprocessor's `#ifdef`-notation [29], makes software difficult to understand [39] and is error prone [28]. 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 [40]. In VDBs, and in the VRA notation for variational 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 variational 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 variational schemas and VDBs, and significant for variational queries written in VRA, despite the fact that such annotations are disciplined. That is, we believe that presence conditions make clear the structural and content variation in our example VDBs without significantly impacting the understandability of the overall structure and content of the variant databases. However, the understandability of variational queries do seem to be significantly impacted by the use of presence conditions and choices, despite the fact that their use is disciplined in the VRA notation.

It is possible that a more restrictive and/or coarse-grained form of variation in variational queries would make them easier to understand at the cost of increased redundancy and (potentially) reduced expressiveness. This tradeoff is one

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

Alternatively, the understandability of variational queries could be improved through tooling, for example, using background colors [27], virtual separation of concerns [36], or view-based editing [59, 53]. Future work should validate our subjective assessment of the understandability VDBs and variational 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 [37]. VDBs and variational 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 [11]. 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 [37]. 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 [56]), 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 [56]). Although not presented here, we build directly on this body of work, especially work on variational typing [16, 17], to enable efficiently checking variational 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.

Chapter 6 Variational Database Management System (VDBMS)

[vdbms] [include example of vq, rqs, sqls] [include sanity checks] add this somewhere: We implemented these checks in our VDBMS tool and verified that both use cases described in this paper satisfy all of them.

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]

from vamos

We proposed encoding variation explicitly in database schemas and queries in [8] and proposed applying this idea to database-backed SPLs in [9]. Our previous work uses slightly different encodings; the one presented here is the basis of our VDBMS implementation. This is the first work that provides use cases for VDBs.

The SPL community has a tradition of developing and distributing use cases to support research on software variation. For example, SPL2go [55] catalogs

the source code and variability models of a large number of SPLs. Additionally, specific projects, such as Apel et al.'s [5] work on SPL verification, often distribute use cases along with study results. However, there are no existing datasets or use cases that include corresponding relational databases and queries, despite their ubiquity in modern software.

Many researchers have recognized the need to manage structural variation in the databases that SPLs rely on. Abo Zaid and De Troyer [2] argue for modeling data variability as part of a model-oriented SPL process. Their *variable data models* link features to concepts in a data model so that specialized data models can be generated for different products. Khedri and Khosravi [38] address data model variability in the context of delta-oriented programming. They define delta modules that can incrementally generate a relational database schema, and so can be used to generate different schemas for each variant of a SPL. Humblet et al. [35] present a tool to manage variation in the schema of a relational database used by a SPL. Their tool enables linking features to elements of a schema, then generating different variants of the schema for different products. Schäler et al. [48] generate a variable database schema from a given global schema and software configurations by mapping schema element to features. Siegmund et al. [50] emphasize the need for variable database schema in SPLs and propose two decomposition approaches: (1) *physical* where database sub-schemas associated with a feature are stored in physical files and (2) *virtual* where a global entity-relation model of a schema is annotated with features. All of these approaches address the issue of *structural* database variation in SPLs and provide a way to derive a schema per variant,

which is also achievable by configuring a VDB. The work of Humblet et al. [35] is most similar to our notion of a variational schema since it is an annotative approach [37] that directly associates schema elements with features. Abo Zaid and De Troyer [2] is also annotative, but operates at the higher level of a data model that may only later be realized as a relational database. Khedri and Khosravi [38] is a compositional approach [37] to generating database schemas. None of these approaches consider *content-level* variation, which is captured by VDBs and observable in our use cases, nor do they consider how to express queries over databases with structural variation, which is addressed by our *v-queries*.

While the previous approaches all address data variation in space, Herrmann et al. [31] emphasize that as an SPL evolves over time, so does its database. Their approach adapts work on database evolution to SPLs, enabling the safe evolution of all deployed products.

Database researchers have studied several kinds of variation in both time and space. There is a substantial body of work on *schema evolution* and *database migration* [20, 43, 32, 45], which corresponds to variation in time. Typically the goal of such work is to safely migrate existing databases forward to new versions of the schema as it evolves. Work on *database versioning* [13, 33] extends this idea to a database's content. In a versioned database, content changes can be sent between different instances of a database, similar to a distributed revision control system. All of this work is different from variational databases because it encodes a less general notion of variation and does not support querying multiple versions of the database at once. Work on *data integration* can be viewed as managing variation in

space [22]. In data integration, the goal is to combine data from disparate sources and provide a unified interface for querying. This is different from VDBs, which make differences between variants explicit.

The representation of v-schemas and variational tables is based on previous work on variational sets [25], which is part of a larger effort toward developing safe and efficient variational data structures [60, 42]. The central motivation of work on variational data structures is that many applications can benefit from maintaining and computing with variation at runtime [24, 18]. The ability to maintain and query several variants of a database at once extends the idea of computing with variation to relational databases.

Chapter 8 Conclusion

[conclusion] from vamos

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

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

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

[7] 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`.

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

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

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

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

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

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

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

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

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

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

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

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

[33] 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.

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

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

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

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

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

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

[40] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Int. Conf. on Aspect-Oriented Software Development*, pages 191–202, 3 2011.

[41] 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`.

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

[43] 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.

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

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

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

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

[52] 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.

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

[54] 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.

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

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

[57] 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.

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

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

[60] 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.