AN ABSTRACT OF THE DISSERTATION OF

Parisa S. Ataei for the degree of Doctor of Philosophy in Computer Science presented on June 11, 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 datasets and scenarios that are partially supported by existing techniques. It presents the variational database management system which is a prototype 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 the 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 with respect to the relational algebra type system and its denotational semantics.

©Copyright by Parisa S. Ataei
June 11, 2021
All Rights Reserved

Theory and Implementation of a Variational Database Management System

by

Parisa S. Ataei

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 11, 2021
Commencement June 2022

Doctor of Philosophy dissertation of Parisa S. Ataei presented on June 11, 2021.

APPROVED:

Major Professor, representing Computer Science

Head 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 S. Ataei, Author

ACKNOWLEDGEMENTS

First and foremost, I am very thankful to my advisor Eric Walkingshaw for piquing my interest in programming languages through our early days of collaboration and functional programming by suggesting that I implement my system in Haskell. Eric's insistence on exploring my ideas and experimenting with them–even when he knew I am heading to the wrong path–allowed me to become more independent in my research over time. His advice and feedback on research and scientific writing taught me lots of valuable lessons. His dedication to making interdisciplinary programming languages research accessible to and understandable by a wide audience through meticulous editing of papers and presentations has been nothing short of exemplary for me. His support and encouragement to tickle my curiosity by taking classes outside of OSU and exploring various resources in our reading group allowed me to find the beautiful corner of programming languages that I absolutely enjoy. His care for my emotional and mental well-being significantly improved the quality of my life throughout my Ph.D. in OSU. Finally, his dedicated, innovative, and caring research and teaching style had a significant role in forming the researcher and (sometimes) the teacher I am today. I am forever grateful for his support, guidance, enthusiasm, and encouragement through the years.

A very special thank you to my partner Jeffrey M. Young for his love, support, and encouragement. Thank you for your technical help–from me learning Haskell when I initially joined the PL group to plotting the diagrams in this thesis. Thank

you for discussing research ideas and problems, practicing many presentations, and studying various books and papers with me. Your dedication to learning, helping others, and community has actively made the PL group at OSU a better place. Thank you for an amazing graduation gift. I look forward to dropping the Ph.D. weight! Finally, thank you for sharing your life with me. You make my life sweeter every day. I look forward to our life together, out of the brutality of graduate school.

I would like to express my gratitude to the many OSU faculty members and peers during my time here. A special thank you to Glencora Borradaile for her unwavering support of graduate students. Also, a very special thank you to Martin Erwig for many constructive advice and discussions on research in either classes or meetings. His dedication to research and teaching greatly influenced the researcher in me. Thank you to the members of my program committee–Martin Erwig, Alan Fern, Amir Nayyeri–for all their support and advice. Thank you to collaborators for professional help and feedback including Arash Termehchy, Qiaoran Li, and Fariba Khan. A very special thank you to my colleagues in the Lambda reading group and the functional programming club for their passion and interesting discussion.

Thank you to other mentors. Thank you to David Christiansen for a great dependent typed languages class at PSU which triggered my enthusiasm to pursue dependent types and formal verification further. It definitely worth the drive! And thank you to Jose Calderson for honest open conversations on building a research career. He is a breathe of fresh air in today's humanity.

Thank you to my strong amazing ladies–Lily Ranjbar, Nasim Adami, Mahtab Aboufazeli, and Saereh Mirzababaei–for their constant kindness, support, and advice during my Ph.D. years. Thank you for always being there for me through the good and the bad. Life has thrown some harsh obstacles my way and I would not have been able to overcome them without each and every one of you.

I would like to thank my friends for check-ins, many conversations, professional and personal help, sharing lots of coffee (sometimes over complaining about grad school), hangouts, and movie nights. Thank you Ali Mohtat, Ali Jafarnejad, Arezoo Rajabi, Reyhaneh Shahmohamadi, Wesley Alexis, Beatrice Moissinac, Mandana Hamidi-Haines, Ralph Haines, Esther and Hamid Mahmoodabadi, Hadi Hashemi, Ben McCamish, Vahid Ghadakchi, Reza Ghaeini, Farzad Zafarani, Mike McGirr, Ameya Ketkar, Colin Shea-Blymyer, and Will Maxwell.

I would also like to thank the CrossFit 97333 community and its coaches. Thank you for providing an environment that allowed me to escape the Ph.D. life even just for a couple of hours a week. Thank you for pushing me physically and mentally through lots of tough, yet, enjoyable workouts.

Last but not least, I am eternally grateful to my parents directing me into this path and prioritizing my education over the years. Thank you for giving me a prospering life and the freedom to grow in the path I desired. I appreciate all your sacrifices and I am deeply sorry for the pain of separation over the last six years. A special thank you to my sister for her love and care. I cannot wait to be reunited with all of you.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

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

Managing variation in databases is a perennial problem in database literature that appears in different forms and contexts [70, 13, 20, 32, 17] and is unavoidable [68]. 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 [53, 19, 9, 67, 56], data integration [28], temporal databases [44, 59, 71], and database versioning [18, 40]. However, there is 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.

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 datasets (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 context of an outline of the structure of the rest of the thesis.

1.1 Motivation and Impact

Managing variation in databases is a perennial problem that appears in different forms and contexts [70, 13, 20, 32, 17]. 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 [53, 19, 9, 67, 56], data integration [28], temporal database [44, 59, 71], and database versioning [18, 40]. 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 (illustrated in Section 1.1.2) and different kinds of variation often interact (illustrated in Section 1.1.1), 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 [53, 19, 9, 67, 56]. 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) [7]. SPL approaches are used 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 [66]. 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 [66, 64, 2]. These solutions can generate a database schema variant for each software variant of the SPL. Unfortunately, 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 [38]. Nor do existing approaches support variation across all levels of a relational database: schema, queries, and content. In this thesis, we argue 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 [10, 11, 13, 12]. 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 thesis using a motivating example described in Section 1.1.1.

To summarize, variation in databases is abundant and inexorable [68] and arises in many different contexts. Existing solutions are specialized to address specific kinds of variation but do not support the interaction of multiple kinds of variation, nor many variation scenarios that cut across different levels of a database. The lack of support for these scenarios negatively impacts database administrators, data analysts, and software developers [38].

1.1.1 Motivating Example

In this section, we motivate VDBMS by illustrating the scenario described in Section 1.1, where two kinds of database variation interact, producing a scenario that is not well-supported by any existing tools. The scenario involves the evolution over time of a database-backed SPL.

An SPL uses a set of boolean variables called *features* to indicate functionality that may be included or not in each software variant. Consider an SPL that

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*)

generates management software for companies. It has a feature *edu* that indicates whether a company provides educational resources, such as courses for its employees. Software variants in which *edu* is enabled (i.e., *edu* = `true`) provide this additional functionality while variants where it is disabled provide only the basic functionality.

If *edu* is the only optional feature, then at any point in time, this SPL has two variants: basic and educational. However, each variant will also evolve as the SPL evolves, leading to several different basic and educational variants over time.

Each variant of the SPL needs a database to store information about employees, and the selection of features impacts the database: While basic variants do not need to store any education-related records, educational variants do. We visualize the impact of both features and evolution on the schema in Table 1.1, where feature variation is captured in the columns and variation over time is captured in the rows. Each basic schema variant contains only the schema in a cell in column basic of Table 1.1a while an educational schema variant consists of two sub-schemas: one from the basic column of Table 1.1a and another from the educational column of Table 1.1b. The basic sub-schema and the educational sub-schema may vary over time independently. For example, an extension to the educational sub-schema may use an older version of the basic schema. This is reflected by the highlighted cells in Table 1.1, which describes a complete schema for a particular educational variant. To describe variation over time of each sub-schema, we introduce two disjoint sets of temporal features (boolean variables, like any other feature), shown in the left columns of Table 1.1a and Table 1.1b.

Now, consider the following scenario: In the initial design of the basic database, the DBA settles 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, combining 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*, they only exist in the basic schema when $V_1 =$ `true`. This scenario is an example of *component evolution* in SPLs, where developers update, refactor, and improve components of the SPL, including the database [38].

Now, consider the case where a client that previously requested a basic variant of the software has added courses to educate its employees in specific subjects. The SPL developer needs to enable the *edu* feature for this client, forcing the adjustment of the schema variant to include the educational sub-schema. This case describes *product evolution*, where database evolution in an SPL results from clients adding/removing features/components [38].

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 exist in the *empacct* relation when *edu* = `true`, represented by a dash-underline, otherwise the *empacct* relation has only the attributes *empno*, *hiredate*, *title*, and *deptno*. Hence, the attributes *std* and *instr* in *empacct* relation are *variational*, that is, they only exist in *empacct* relation when *edu* = `true`.

Our example demonstrates how different kinds of variation interact with each other, an inevitable 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 datasets (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 performances.

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

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

Chapter 3 (*Variational Database Framework*) describes a formal model of *variational databases* (VDBs), where the structure of data is defined by a *variational schema* (Section 3.1) and the content is defined by *variational tables* (Section 3.2).

In essence, a variational database gathers multiple relational databases in one place. Section 3.1 and Section 3.2 also describe how a relational database can be generated from a VDB, that is, how a variational database can be deployed to a relational database for a variant. Finally, Section 3.3 defines properties of a well-formed VDB.

Chapter 4 (*Variational Queries*) describes the need for a query language to extract information from a VDB. It formally defines *variational relational algebra* (VRA) as a query language for VDB (Section 4.1). It also describes an *explicitly annotating* function for queries to relieve the user from repeating the VDB's variation in their queries in Section 4.3. Additionally, it describes a *static type system* for ensuring that all variants of a query are compatible with the corresponding variants of the VDB (Section 4.2). Furthermore, it defines the denotational semantics of VRA through the semantics of relational algebra (Section 4.4). It also defines a set of syntax-based rules to *minimize variation* in variational queries (Section 4.5). Finally, it discusses the properties of the VRA in Section 4.6 including its expressiveness and type safety. This chapter heavily borrows from our papers [10, 11, 12].

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, database variants do not exist. Thus, generating the VDB requires the expert knowledge and cannot be automated. It details two such variation scenarios and introduces two use cases of VDB, one over space (adapted from the email SPL described by Hall [37] and explained in Section 5.1) and another over

time (adapted from the evolution of an employee schema described by Moon et al. [56] and explained in Section 5.2). Additionally, it describes 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?".

Chapter 5 heavily borrows from our previous paper [13] and Qiaoran Li's Masters project report [51]. Although the databases and queries were originally taken from Qiaoran's work, they have been significantly modified so that the VDBs pass the properties of a well-defined VDB and the queries justify the information need of their scenario better.

Chapter 6 (*Variational Database Management System*) discusses how we implemented the variational database framework and the variational relational algebra into a Variational Database Management System (VDBMS) on top of a traditional RDBMS. This chapter includes the architecture of VDBMS and approaches used to generate SQL queries to run variational queries on a backend RDBMS. It also compares the performance of VDBMS using different SQL generator approaches.

Chapter 7 (*Related Work*) describes previous research on different kinds of variation in databases.

Finally, Chapter 8 (*Conclusion*) briefly summarizes the main contributions of this thesis and immediate future works.

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, 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 $i \in \mathbf{Int}$ to state that the variable i is of type **Int**. Types can be more general. Consider the type **Set a** that indicates the set of all sets of values of type **a**. Here, **a** is a 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 to make formalizations easier to understand. For example, instead of referring to **Set Int** we can give it a new name (**Ints** = **Set Int**). Thus, **Ints** also indicates all possible sets of integers. Sometimes we must extend a type with an additional "bottom" element (i.e., $\perp$). To account for this extension at the type level we subscript the type with $\perp$. For example, $\mathbf{Int}_\perp$ denotes the extension of type **Int** with $\perp$. Furthermore, sometimes

we pair two types together. For 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 type signature $id : \mathbf{a} \rightarrow \mathbf{a}$ for function id states that the function id takes a value of type **a** and returns a value of type **a**.

Throughout the thesis, we discuss relational concepts and their variational counterparts. 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 the variational database framework and how we describe parts of that space using propositional formulas of boolean variables [11, 10]. 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 [31, 77, 10] and Section 2.5 describes the formula choice calculus used to incorporate variation into relational algebra [41].

2.1 Relational Databases

In this section, we introduce relational database concepts, which are the basis of variational databases. Table 2.1 represents an example relational database instance corresponding to V_2, the highlighted cell 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, and 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, the implementation of our database system 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:

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

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 $\leq_{\mathbf{AttrName}}$ in all relations.

Figure 2.1 defines the relational database objects and type synonyms. 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 corresponds 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 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

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 variational 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.

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

(a) Result of the query $\underline{q}_1 = \sigma_{salary \geq 65000}(job)$.

result

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

(b) Result of the query $\underline{q}_2 = (\sigma_{salary \geq 65000}(job)) \times empacct$.

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 $\underline{q}_3 = \sigma_{empacct.title=job.title}((\sigma_{salary \geq 65000}(job)) \times empacct)$.

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 $\underline{q}_4 = \pi_{name,salary}(\sigma_{empacct.title=job.title}((\sigma_{salary \geq 65000}(job)) \times empacct))$.

result

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

The relational algebra that we use also includes standard set operations, a join 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 sets of tuples returned by the subqueries. The *join* operation is

Operators:

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

Constant:

$k \in$ **AttrDom**

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.

equivalent to selection applied to a 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 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

Feature expression syntax:

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

Evaluation of feature expressions:

$$
\begin{aligned}
\mathbb{E}[\![.]\!] &: \textbf{FeatExpr} \rightarrow \textbf{Config} \rightarrow \textbf{Bool} \\
\mathbb{E}[\![b]\!]_c &= b \\
\mathbb{E}[\![f]\!]_c &= c\ f \\
\mathbb{E}[\![\neg e]\!]_c &= \begin{cases} \texttt{false}, & \text{if } \mathbb{E}[\![e]\!]_c = \texttt{true} \\ \texttt{true}, & \text{otherwise} \end{cases} \\
\mathbb{E}[\![e_1 \wedge e_2]\!]_c &= \begin{cases} \texttt{true}, & \text{if } \mathbb{E}[\![e_1]\!]_c = \texttt{true}\ \textit{and}\ \mathbb{E}[\![e_2]\!]_c = \texttt{true} \\ \texttt{false}, & \text{otherwise} \end{cases} \\
\mathbb{E}[\![e_1 \vee e_2]\!]_c &= \begin{cases} \texttt{true}, & \text{if } \mathbb{E}[\![e_1]\!]_c = \texttt{true}\ \textit{or}\ \mathbb{E}[\![e_2]\!]_c = \texttt{true} \\ \texttt{false}, & \text{otherwise} \end{cases}
\end{aligned}
$$

Relations over feature expressions:

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

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

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. The feature space is captured by a closed set of features F.

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* ($\equiv$) on feature expressions corresponding to logical equivalence and 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 $F = \{f_1, f_2\}$, 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)** [31, 77, 10]. 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 *configuring* a variational set with a configuration. This is done by the *variational*

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, the left-hand side $\cup$ denotes the union of variational sets while the right-hand side one denotes the union of plain sets.

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.

Example 2.4.1. Assume we have the feature space $F = \{f_1, f_2\}$ 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, $f_1 = \texttt{true}, f_2 = \texttt{true}$) 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 disjunct 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 4.2. 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 [41] which is an extension of the choice calculus [75, 29]. The choice calculus is a metalanguage for describing variation in programs and its elements such as data structures [77, 31]. 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 [48].

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 [41]. 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

In this section, we define how variation is encoded at the schema level. We first present an example that illustrates how relational databases fail to encode variation at the schema level and how we can express variation in 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 variational 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 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 expression $V_3 \vee V_4 \vee V_5$ indicates that the *empbio* table is present for variants that enable one of V_3–V_5. The three attributes *empno*, *sex*, and *birthdate* are present in all variants where the *empbio* relation exists, so their presence conditions 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 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 focuse only on the relation schema and does 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. Note that data from earlier variants like $\{V_3\}$ is propagated to the later variants like $\{V_4\}$ and $\{V_5\}$.

(a) The relational table of *empbio* for variants that only enables 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 enables the feature V_4 out of the features V_1–V_5.

empbio

empno	*sex*	*birthdate*	*name*
12001	F	1960-11-06	Ulf Hofstetter
12002	M	1961-04-15	Luise McFarlan
12003	M	1958-07-27	Shir DuCasse
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 enables the feature V_5 out of the features V_1–V_5.

empbio

empno	*sex*	*birthdate*	*firstname*	*lastname*
12001	F	1960-11-06	Ulf	Hofstetter
12002	M	1961-04-15	Luise	McFarlan
12003	M	1958-07-27	Shir	DuCasse
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
...	...	...	...	...

(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.

$V_3 \vee V_4 \vee V_5$	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	Ulf Hofstetter	Ulf	Hofstetter
	12002	M	1961-04-15	Luise McFarlan	Luise	McFarlan
	12003	M	1958-07-27	Shir DuCasse	Shir	DuCasse
	80001	M	1956-09-30	Nagui Merli	Nagui	Merli
	80002	M	1963-04-25	Mayuko Meszaros	Mayuko	Meszaros
	80003	F	1960-10-26	Theirry Viele	Theirry	Viele
	200001	M	1960-01-11	Selwyn Koshiba	Selwyn	Koshiba
	200002	M	1957-09-10	Bedrich Markovitch	Bedrich	Markovitch
	200003	F	1961-02-07	Pascal Benzmuller	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	Ulf Hofstetter	Ulf	Hofstetter
	12002	M	1961-04-15	Luise McFarlan	Luise	McFarlan
	12003	M	1958-07-27	Shir DuCasse	Shir	DuCasse
	80001	M	1956-09-30	Nagui Merli	Nagui	Merli
	80002	M	1963-04-25	Mayuko Meszaros	Mayuko	Meszaros
	80003	F	1960-10-26	Theirry Viele	Theirry	Viele
	200001	M	1960-01-11	Selwyn Koshiba	Selwyn	Koshiba
	200002	M	1957-09-10	Bedrich Markovitch	Bedrich	Markovitch
	200003	F	1961-02-07	Pascal Benzmuller	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 $F = \{V_4, V_5, edu, T_4, T_5\}$. 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 $V_5 = \texttt{true}$.

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

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 $pc(r, S) = e_r \wedge pc(S)$. For example, in Example 3.1.1 we have $pc(empacct, S_1) = (V_4 \vee V_5) \wedge e_1$. Further-

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.

more, 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 $F = \{edu, V_1, V_2, V_3, V_4, V_5, T_1, T_2, T_3, T_4, T_5\}$. Additionally, all schema variants are illustrated inTable 1.1.

The feature model e_{mot} only allows one temporal feature to be true from a set of temporal features at the time.

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

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.

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

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

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

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 conditions can change, since different feature expressions can 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 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, $u \in \mathbf{Tuple} \coloneqq (v_1, \ldots, v_l)^e$. A variational tuple corresponds to a variational relation,

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	Ulf Hofstetter	Ulf	Hofstetter	$V_3 \vee V_4 \vee V_5$
	12002	M	1961-04-15	Luise McFarlan	Luise	McFarlan	$V_3 \vee V_4 \vee V_5$
	12003	M	1958-07-27	Shir DuCasse	Shir	DuCasse	$V_3 \vee V_4 \vee V_5$
	80001	M	1956-09-30	Nagui Merli	Nagui	Merli	$V_4 \vee V_5$
	80002	M	1963-04-25	Mayuko Meszaros	Mayuko	Meszaros	$V_4 \vee V_5$
	80003	F	1960-10-26	Theirry Viele	Theirry	Viele	$V_4 \vee V_5$
	200001	M	1960-01-11	Selwyn Koshiba	Selwyn	Koshiba	V_5
	200002	M	1957-09-10	Bedrich Markovitch	Bedrich	Markovitch	V_5
	200003	F	1961-02-07	Pascal Benzmuller	Pascal	Benzmuller	V_5
	...	...	...	...	...	...	...

$r(a_1, \ldots, a_l)^{e_r}$, 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, $U \in \mathbf{RelCont} \coloneqq \{u_1, \ldots, u_k\}$ 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, $\mathcal{I} \in \mathbf{DBInst} \coloneqq \{t_1, \ldots, t_n\}^e$. 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 [13]. Section 3.3 expands on this.

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,

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:

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

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:

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

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.

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 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 [40], the schema level cannot. For example, $\textit{ecourse}\left(\textit{courseno}, \textit{coursename}, \textit{deptno}^{T_5}\right)^{edu \wedge (T_4 \vee T_5)}$ 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.

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 decision 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:

   $\forall u \in r, \forall a \in r, \forall r \in S.\neg sat(pc(r, S) \wedge pc(a, r) \wedge pc(u, r)) \Rightarrow value_u(a) =$ `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 dataset, introduced in Section 5.2. To check this, assume that configurations $c_1, c_2, \cdots$ represent time-ordered configurations, then check $\forall c_i, c_j \in \textbf{Config}, i \leq j, \mathbb{I}[\![\mathcal{I}]\!]_{c_i} \subseteq \mathbb{I}[\![\mathcal{I}]\!]_{c_j}$, 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

Now that we have introduced the variational database framework we need a query language to extract information from a VDB instance. Our approach will build on existing relational query languages (like SQL and relational algebra) but must also account for the new aspect of our database: variation.

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.

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 4.2. 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 4.3.

To understand the meaning of variational queries we define the semantics of variational queries via the semantics of relational queries in Section 4.4. We define how to configure a variational query to a relational query in Section 4.4.1. Then,

we use the results of multiple relational queries to accumulate the result of the original variational query in Section 4.4.2.

We also provide a set of syntactic rules that are semantic-preserving in Section 4.5. These rules enable factoring and distributing variation points within a variational query, which enables syntactic refactoring including maximizing sharing within a variational query. Finally, in Section 4.6, we present some properties of the VRA including the expressiveness and type safety of VRA in Section 4.6.1 and Section 4.6.2, respectively, in addition to the *variation-preservation* property of VRA at the semantics level in Section 4.6.3.

4.1 Variational Relational Algebra

To account for variation, VRA combines relational algebra (RA) with *choices* [29, 41, 75]. 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

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.

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 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 $F = \{V_3, V_4, V_5\}$, 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:

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

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.

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

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 4.1 shows the result of query q_0 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 4.2.

Table 4.1: Result of the v-query $q_0 = \pi_{empno,name,firstname,lastname}(empbio)$.

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

Now we pay attention to the variational aspect of the query. Knowing that the variation encoded in the VDB can be inferred (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.

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

Table 4.2 shows the result of this query over the described VDB. Note that the first three tuples from Table 4.1 are not returned since the query does not project the *empno* attribute for variants that enable V_3 and attributes *name*, *firstname*, and *lastname* do not exist for these variants in the VDB. Thus, the tuple will just be empty and so is dropped.

Table 4.2: Result of the v-queries $q_1 = \pi_{empno^{V_4 \vee V_5}, name, firstname, lastname}(empbio)$ and $q_1' = \pi_{empno^{(V_4 \vee V_5) \wedge \neg V_3}, name^{V_4 \wedge \neg V_3 \wedge \neg V_5}, firstname^{V_5 \wedge \neg V_3 \wedge \neg V_4}, lastname^{V_5 \wedge \neg V_3 \wedge \neg V_4}}(empbio)$.

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*
	12001	Ulf Hofstetter	Ulf	Hofstetter	$V_3 \vee V_4 \vee V_5$
	12002	Luise McFarlan	Luise	McFarlan	$V_3 \vee V_4 \vee V_5$
	12003	Shir DuCasse	Shir	DuCasse	$V_3 \vee V_4 \vee V_5$
	80001	Nagui Merli	Nagui	Merli	$V_4 \vee V_5$
	80002	Mayuko Meszaros	Mayuko	Meszaros	$V_4 \vee V_5$
	80003	Theirry Viele	Theirry	Viele	$V_4 \vee V_5$
	200001	Selwyn Koshiba	Selwyn	Koshiba	V_5
	200002	Bedrich Markovitch	Bedrich	Markovitch	V_5
	200003	Pascal Benzmuller	Pascal	Benzmuller	V_5
	...	...	...	...	...

If desired, we can also make the inferred presence conditions explicit, as demonstrated in the following query q_1'.

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

The result of the query q_1' is still Table 4.2. Note that all the variation encoded in the VDB is applied to the result of a query. Thus, the result of a variational query stands on its own, that is, it is not part of a bigger structure like the variational

tables in a VDB.

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 4.3. q_1 queries all three variants simultaneously although the returned results are only associated with variants V_4 and V_5 due to the annotation of the attribute *empno* in the query and the presence conditions of the rest of the projected attributes in the schema. Yet, the query can be further simplified with a choice. q_2 selects only two out of the three variants explicitly:

$$q_2 = \neg V_3\langle \pi_{empno,name,firstname,lastname}(empbio), \varepsilon \rangle \quad .$$

Table 4.3 shows the result of this query over the VDB described in Example 4.1.1.

Table 4.3: Result of the v-query $q_2 = \neg V_3\langle \pi_{empno,name,firstname,lastname}(empbio), \varepsilon \rangle$.

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*
	12001	Ulf Hofstetter	Ulf	Hofstetter	$V_4 \vee V_5$
	12002	Luise McFarlan	Luise	McFarlan	$V_4 \vee V_5$
	12003	Shir DuCasse	Shir	DuCasse	$V_4 \vee V_5$
	80001	Nagui Merli	Nagui	Merli	$V_4 \vee V_5$
	80002	Mayuko Meszaros	Mayuko	Meszaros	$V_4 \vee V_5$
	80003	Theirry Viele	Theirry	Viele	$V_4 \vee V_5$
	200001	Selwyn Koshiba	Selwyn	Koshiba	V_5
	200002	Bedrich Markovitch	Bedrich	Markovitch	V_5
	200003	Pascal Benzmuller	Pascal	Benzmuller	V_5
	...	...	...	...	...

Note that, as shown in Table 4.2 and Table 4.3, queries q_1 and q_2 return the same set of variational tuples. However, the first three tuples in Table 4.2 could belong to a variant that enables any of V_3–V_5 whereas the first three tuples in Table 4.3 could only belong to variants that either enable V_4 or V_5. This difference

is due to the difference in their tables' presence conditions, that is, 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 4.2.1. More importantly, even though the first three tuples in Table 4.2 could belong to a variant that enables V_3, configuring Table 4.2 for such a variant drops the first three tuples, since all their attributes would be `NULL`. We illustrate how configuring Table 4.2 for variant $\{V_3\}$ drops the first three tuples in Example 4.4.1.

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 $F = \{V_1, \ldots, V_5\}$ 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 Type System

In this section, we introduce a static type system for VRA. The type system ensures that queries are consistent with the underlying variationalschema. That is, that all referenced relations and attributes are present in the variation contexts in which they are used. For example, consider the VDB from Example 4.4.1 that contains only the relation $r({a_1}^{f_1}, a_2, a_3)^{f_1 \vee f_2}$. The query $\pi_{a_4} r$ is ill-typed since a_4 is not present in r. Similarly, the queries $\pi_{{a_1}^{\neg f_1}} r$ and $f_1\langle \pi_{a_2} r, \pi_{a_1} r \rangle$ are both ill-typed since a_1 is not present in r when f_1 is disabled.

The type of a VRA query is a variational relation schema $\mathit{result}(A)^e$. However, since the relation name is the same for all queries, we shorten this to A^e, that is, an annotated variational set of attributes. The annotation e corresponds to the presence condition of the returned table. The presence conditions of attributes within A may differ from the corresponding presence conditions in the original variational schema due to variation constraints imposed by the query. For example, continuing with relation $s = r({a_1}^{f_1}, a_2, a_3)^{f_1 \vee f_2}$, the query $\pi_{{a_2}^{f_1}} r$ has type $\{{a_2}^{f_1}\}^{f_1 \vee f_2}$. In the original schema, a_2 is present when $f_1 \vee f_2$, while in the query it is present only

when f_1 is enabled.

Figure 4.2 defines a typing relation that relates VRA queries to their types. The judgment form $e, S \vdash q : A^{e'}$ states that in variation context e within variational schema S, variational query q has type $A^{e'}$. If a query does not have a type, it is *ill-typed*. A *variation context* is a feature expression that tracks which variants the current subquery is present in. We sometimes use the judgment form $S \vdash q : A^{e'}$ when the variation context is the unextended feature model, that is, $pc(S), S \vdash q : A^{e'}$. We assume that the variational set of attributes A is normalized to remove elements with unsatisfiable presence conditions, but this normalization is only shown explicitly in the rules where strictly necessary.

The rule Relation-E looks up relation r in the variational schema S and returns its variational set of attributes A. The presence condition of A is the conjunction of the relation's presence condition in the variational schema, e', the current variation context, e, and the feature model, $pc(S)$. In this way, the type is constrained to reflect both the constraints present in the variational schema and the context of the relation reference in the query. The last premise ensures that the relation exists in at least one variant by checking that the type's presence condition is satisfiable. This means that referencing a relation in a context where it is never present is a type error.

For a projection $\pi_A q$, the rule Project-E checks that all projected attributes A are present in at least one variant of the variation context (second premise) and that these attributes are *subsumed* by type of the subquery q (third premise). The subsumption relation $A \prec A'$ used in the third premise is defined as $\forall a^{e_1} \in \downarrow$

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 4.2: 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.

$(A)\,.\exists e_2.a^{e_2} \in \downarrow(A')\,, sat(e_1 \wedge e_2)$, which ensures that all of the projected attributes are present in the type of the subquery q, and that the presence conditions of the variational set of projected attributes do not contradict the presence conditions in the type of q. The result type is the variational set intersection (Figure 2.4) of the projected attributes and the attributes of the subquery ensuring that the variation constraints of both are captured. Example 4.2.1 illustrates how the type system infers a type for a variational query.

The rule SELECT-E checks if its subquery and variational condition are well-typed and if so it returns the subquery's type. The variational condition typing relation is defined in Figure 4.2 and has the judgment form $e, A \vdash \theta$, which states that the variational condition θ is well-formed in variation context e within attribute variational set A. The variational condition typing rules ensure that each attribute used in a variational condition is present in A and that the presence condition associate with that attribute does not contradict the current variation context.

For a choice of queries $e'\langle q_1, q_2\rangle$, the rule CHOICE-E recursively infers the type of each alternative subquery in a variation context extended to reflect which branch of the choice the subquery is contained in, that is, e' for q_1 and $\neg e'$ for q_2. The result type of a choice is the variational set union (Figure 2.4) of the types of the subqueries annotated by the disjunction of their presence conditions, reflecting that either one alternative will be chosen or the other. Note that CHOICE-E is the only rule that refines the variation context.

The EMPTYRELATION-E rule states that an empty relation has the type of an

empty set annotated by `false`, which is the required base case to ensure that the type system is variation preserving (see Section 4.6.2). The remaining rules are straightforward extensions of the standard relational algebra typing rules for product and set operations to account for variation contexts and variational sets.

Example 4.2.1. Consider the query q_1 given in Example 4.1.1. Through this example, we simplify feature expressions when possible. First, the PROJECT-E rule is applied which requires the three premises to hold: The first one looks up the type of *empbio* relation from the underlying variational schema (Assumption 1 in Figure 4.3). The second one checks for the subsumption of the projected attributes from the type of the subquery (Assumption 2 in Figure 4.3). The last one ensures all projected attributes are valid under the current variation context (Assumption 3 in Figure 4.3). Since all premises hold the type of the query is generated as shown in Figure 4.3:

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

Now consider q_2 introduced in Example 4.1.1. First, the CHOICE-E rule is applied which requires two premises to hold: The first one checks that the left alternative (which then applies the PROJECT-E rule) of the choice is type-correct, in which case it also generates its type (Assumption 5 in Figure 4.4). The second one does the same for the right alternative which is an empty relation query and is always type-correct. Since both alternative are type-correct the type of the query

Figure 4.3: Derivation tree for q_1 in Example 4.2.1.

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:

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

is generated as shown in Figure 4.4:

$$(empno, name^{V_4}, firstname^{V_5}, lastname^{V_5})^{(e_2 \wedge (\neg V_3)) \vee \texttt{false}}$$

Figure 4.4: Derivation tree for q_2 in Example 4.2.1.

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:

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

4.3 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:

$q_6 = \pi_{empno^{(V_4 \vee V_5) \wedge \neg V_3}, name^{\neg V_3 \wedge V_4 \wedge \neg V_5}, firstname^{\neg V_3 \wedge \neg V_4 \wedge V_5}, lastname^{\neg V_3 \wedge \neg V_4 \wedge V_5}} (e_2\langle empbio, \varepsilon \rangle)$.[1]

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. 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 4.5, 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,

[1] 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 $\lfloor empbio \rfloor_{S_2} = e_2\langle \pi_{empno, name^{V_4}, firstname^{V_5}, lastname^{V_5}}(empbio), \varepsilon \rangle$.

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

Figure 4.5: Explicitly annotating a well-typed query with a variational schema.

after simplification [2].

Theorem 4.3.1. If the query q has the type A^e then its explicitly annotated counterpart has an equivalent type, that is:

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

Proof. By structural induction. We encoded and proved this theorem in the Coq proof assistant [48]. □

This theorem shows that the type system applies the schema to the type of a query although it 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 illustrate the application of Theorem 4.3.1 to queries q_1 and q_6. Example 4.2.1 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.

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

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, $A_{q_1} \equiv A_{q_6}$.

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 which is explained in Section 4.4.1. This is contra to other functions that have to take both the query and the schema, such as the type system. We explain this in more details in Section 4.4.1.

4.4 VRA Semantics

We use the semantics of relational queries to define the semantics of variational queries. In Section 4.4.1, we 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 query variants. Then, in Section 4.4.2, we define an accumulation function that accumulates multiple (annotated) relational tables into a variational table. Finally, in Section 4.4.3, we define the denotational semantics of VRA using the defined configuration and accumulation functions.

4.4.1 VRA Configuration

The *configuration* function maps an explicitly annotated variational query under a configuration to a relational query, defined in Figure 4.6. Thus, a variational query can be understood as a set of relational queries, the results of which are gathered in a single table and tagged with the feature expression stating their variants. Users can deploy queries for a specific variant by configuring the variational query. Note that the configuration function takes well-typed, explicitly annotated queries. Example 4.4.1 illustrates configuring a query and explains why a query passed to the configuration function must be explicitly annotated. Example 4.4.2 illustrates the configuration of query q_1' from Example 4.1.1 and the corresponding relational results table.

Example 4.4.1. 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 $F = \{f_1, f_2\}$. For valid configurations of this VDB (that is, $\{\ \}$, $\{f_1\}$, $\{f_2\}$, and $\{f_1, f_2\}$), 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:

$$\mathbb{Q}[\![q_5]\!]_{\{\ \}} = \pi_{a_1}(\underline{r})$$

$$\mathbb{Q}[\![q_5]\!]_{\{f_1\}} = \pi_{a_1}(\underline{r})$$

$$\mathbb{Q}[\![q_5]\!]_{\{f_2\}} = \pi_{a_1, a_3}(\underline{r})$$

$$\mathbb{Q}[\![q_5]\!]_{\{f_1, f_2\}} = \pi_{a_1, a_2, a_3}(\underline{r})$$

However, the query q_5 is not explicitly annotated since attribute a_1 does not carry

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:

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

Figure 4.6: Configuration of variational queries and conditions. The configuration function assumes that the input, either the variational query or the variational condition, is well-typed. Additionally, the query passed to the configuration function must be explicitly annotated.

its variational encoding from the database, that is, it does not have the presence condition f_1. Explicitly annotating this query gives us query $q_5' = \lfloor q_5 \rfloor_{S_3} = \pi_{a_1{}^{f_1}, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2}}(r)$. Configuring q_5' results in the same query as configuring q_5 except for configurations $\{\ \}$ and $\{f_1\}$. Thus, the correct configuration of q_5 is:

$$\mathbb{Q}[\![\lfloor q_5 \rfloor_{S_3}]\!]_{\{\ \}} = \varepsilon$$
$$\mathbb{Q}[\![\lfloor q_5 \rfloor_{S_3}]\!]_{\{f_1\}} = \pi_{a_1}(\underline{r})$$
$$\mathbb{Q}[\![\lfloor q_5 \rfloor_{S_3}]\!]_{\{f_2\}} = \pi_{a_3}(\underline{r})$$
$$\mathbb{Q}[\![\lfloor q_5 \rfloor_{S_3}]\!]_{\{f_1,f_2\}} = \pi_{a_1,a_2,a_3}(\underline{r})$$

The reason why $\mathbb{Q}[\![q_5]\!]_{\{\ \}}$ and $\mathbb{Q}[\![q_5]\!]_{\{f_1\}}$ are 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.

Example 4.4.2. Consider the query q_1' given in Example 4.1.1 which is already explicitly annotated:

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

Configuring q_1' for all valid configurations ($\{V_3\}$, $\{V_4\}$, $\{V_5\}$) of the given VDB

results in three relational queries:

$$\mathbb{Q}[\![q_1']\!]_{\{V_3\}} = \varepsilon$$
$$\mathbb{Q}[\![q_1']\!]_{\{V_4\}} = \pi_{empno,name}(empbio)$$
$$\mathbb{Q}[\![q_1']\!]_{\{V_5\}} = \pi_{empno,firstname,lastname}(empbio)$$

Table 4.4 shows the result of these relational queries.

Often a variational query will yield the same plain query for multiple configurations. For our semantics, it is useful to get the set of unique variants of a variational query. Thus, we define the *unique variants* (unique configuration) function, whose type is given below.

$$\mathcal{Q}(\cdot,\cdot) : \mathbf{Q} \rightarrow \mathbf{Set\ FeatName} \rightarrow \mathbf{Set\ (Var\ \underline{Q})}$$

This function takes a well-typed, explicitly annotated variational query and VDB's set of features 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. To generate this presence condition from configurations we need to know the closed set of VDB's features. This is done by the $genFexp(c, F)$ that takes a configuration and a closed set of features and generates the feature expression e that is only satisfiable by the configuration c. For example, $genFexp(\{f_1\}, \{f_1, f_2\}) = f_1 \wedge \neg f_2$ and $genFexp(\{f_1, f_2\}, \{f_1, f_2\}) = f_1 \wedge f_2$. Re-

Table 4.4: 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*
	12001	Ulf Hofstetter
	12002	Luise McFarlan
	12003	Shir DuCasse
	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*
	12001	Ulf	Hofstetter
	12002	Luise	McFarlan
	12003	Shir	DuCasse
	80001	Nagui	Merli
	80002	Mayuko	Meszaros
	80003	Theirry	Viele
	200001	Selwyn	Koshiba
	200002	Bedrich	Markovitch
	200003	Pascal	Benzmuller
	...	...	...

member that the set of enabled features of a configuration denote the said configuration, for example, $\{f_1\}$ denotes the configuration in which only feature f_1 has been enabled.

In essence, the unique variants function can be defined for all data types that encode variation. For example, the unique configuration function for variational queries can be defined as follows.

$$\mathcal{Q}(q, F) = \{\underline{q}^{e_1 \vee \ldots \vee e_n} \mid \underline{q}^{e_1}, \ldots, \underline{q}^{e_n} \in \{(\mathbb{Q}[\![q]\!]_c)^{genFexp(c,F)} \mid c \in \mathbf{Config}\}\}$$

The unique configuration for variational sets of attributes ($\mathcal{A}(\cdot,\cdot)$) and variational conditions ($\mathcal{C}(\cdot,\cdot)$) are defined similarly; their types are given below.

$$\mathcal{A}(\cdot,\cdot) : \mathbf{Set\ (Var\ AttrName)} \rightarrow \mathbf{Set\ FeatName} \rightarrow \mathbf{Var\ (Set\ AttrName)}$$
$$\mathcal{C}(\cdot,\cdot) : \mathbf{Condition} \rightarrow \mathbf{Set\ FeatName} \rightarrow \mathbf{Var\ \underline{Condition}}$$

However, the definition of $\mathcal{Q}(\cdot,\cdot)$ is not efficient, since it still enumerates all possible configurations. Thus, we define the more efficient unique configuration function for variational queries in Figure 4.7.

Example 4.4.3. Consider the variational query $q_5' = \pi_{a_1{}^{f_1}, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2}}(r)$ given in Example 4.4.1, which is well-typed and explicitly annotated. The unique configuration of this query results in the following set of queries:

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

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

Figure 4.7: Unique configuration of variational queries. The unique configuration function assumes that the input variational query is well-typed and explicitly annotated by the underlying variational schema of the VDB.

Example 4.4.4. Consider the query q_1' configured in Example 4.4.2:

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

The unique configuration of it results in:

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

4.4.2 Accumulation of Relational Tables to a Variational Table

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 → Set (Config, Table) → Table** takes the feature space of a database and a set of relational tables with their attached configurations and generates a variational table. Figure 4.8 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 adjusts the content of tables by mapping 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 attached configuration given the closed set of features of the database's feature space.[3] Note that to

[3]In the 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.

Table accumulation function:

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

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 4.8: 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.

generate a feature expression from a configuration it is essential to pass the closed set of features. Example 4.4.5 illustrates the behavior of these auxiliary functions and the table accumulation function over the relational tables in Table 4.4.

Example 4.4.5. Consider the query q_1' written over the VDB with variational schema S_2 and feature space $F = \{V_3, V_4, V_5\}$, all given in Example 4.1.1. All configured relational queries of q_1' for VDB's valid configurations and their corresponding results in form of a relational table are given in Example 4.4.2 and

Table 4.4, respectively. Now we show how the relational tables of the configured queries, shown in Table 4.4, are accumulated to the variational table, shown in Table 4.2, 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 4.4. 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 4.4b) and $\{V_5\}$ (associated to Table 4.4c), 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 4.4 are adjusted so that they all match a certain relation schema. This is achieved by the *fitConfTablesToVsch* which gets all the tables in Table 4.4 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 4.4 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 (indicated by the white space) in the tuples for those attributes. Table 4.5–Table 4.7 illustrate the application of *fitTableToVsch* to Table 4.4a–Table 4.4c and variational relation schema s_{accum}.

Table 4.5: Result of the *fitTableToVsch* applied to Table 4.4a and variational relation schema s_{accum}.

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

Table 4.6: Result of the *fitTableToVsch* applied to Table 4.4b and variational relation schema s_{accum}.

result	*empno*	*name*	*firstname*	*lastname*
	12001	Ulf Hofstetter		
	12002	Luise McFarlan		
	12003	Shir DuCasse		
	80001	Nagui Merli		
	80002	Mayuko Meszaros		
	80003	Theirry Viele		
	...	...	...	...

Then, the *addPresCondToConfContent* function adds the presence condition attribute and its values to relation contents of Table 4.5–Table 4.7, resulting in Table 4.8 which illustrates a set of relation contents that are separated by the red bold line. Note that since Table 4.5 does not have any tuples, Table 4.8 does not have any tuples associated with the variant $\{V_3\}$.

Table 4.7: Result of the *fitTableToVsch* applied to Table 4.4c and variational relation schema s_{accum}.

result

empno	*name*	*firstname*	*lastname*
12001		Ulf	Hofstetter
12002		Luise	McFarlan
12003		Shir	DuCasse
80001		Nagui	Merli
80002		Mayuko	Meszaros
80003		Theirry	Viele
200001		Selwyn	Koshiba
200002		Bedrich	Markovitch
200003		Pascal	Benzmuller
...	...	...	...

Table 4.8: 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.

12001	Ulf Hofstetter			$\neg V_3 \wedge V_4 \wedge \neg V_5$
12002	Luise McFarlan			$\neg V_3 \wedge V_4 \wedge \neg V_5$
12003	Shir DuCasse			$\neg V_3 \wedge V_4 \wedge \neg V_5$
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			$\neg V_3 \wedge V_4 \wedge \neg V_5$
...	...	...	...	...
12001		Ulf	Hofstetter	$\neg V_3 \wedge \neg V_4 \wedge V_5$
12002		Luise	McFarlan	$\neg V_3 \wedge \neg V_4 \wedge V_5$
12003		Shir	DuCasse	$\neg V_3 \wedge \neg V_4 \wedge V_5$
80001		Nagui	Merli	$\neg V_3 \wedge \neg V_4 \wedge V_5$
80002		Mayuko	Meszaros	$\neg V_3 \wedge \neg V_4 \wedge V_5$
80003		Theirry	Viele	$\neg V_3 \wedge \neg V_4 \wedge V_5$
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	$\neg V_3 \wedge \neg V_4 \wedge V_5$
...	...	...	...	...

Table 4.9: Final step of table accumulation passes the variational relation schema s_{accum} and relation contents in Table 4.8 to the *mkTable* function.

oneof (V_3, V_4, V_5)	$(\neg V_3 \wedge V_4 \wedge \neg V_5)$ $(\vee(\neg V_3 \wedge \neg V_4 \wedge V_5)$	$\neg V_3 \wedge V_4 \wedge \neg V_5$	$\neg V_3 \wedge \neg V_4 \wedge V_5$	$\neg V_3 \wedge \neg V_4 \wedge V_5$	`true`
result	*empno*	*name*	*firstname*	*lastname*	*prescond*
	12001	Ulf Hofstetter			$\neg V_3 \wedge V_4 \wedge \neg V_5$
	12002	Luise McFarlan			$\neg V_3 \wedge V_4 \wedge \neg V_5$
	12003	Shir DuCasse			$\neg V_3 \wedge V_4 \wedge \neg V_5$
	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			$\neg V_3 \wedge V_4 \wedge \neg V_5$
	12001		Ulf	Hofstetter	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	12002		Luise	McFarlan	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	12003		Shir	DuCasse	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	80001		Nagui	Merli	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	80002		Mayuko	Meszaros	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	80003		Theirry	Viele	$\neg V_3 \wedge \neg V_4 \wedge V_5$
	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 4.8. Note that the values in tuples of Table 4.8 follow the order of the attributes in the variational relation schema. This results in Table 4.9 which is equivalent to the result of q_1' given in Table 4.2.

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 4.9 defines this function and its auxiliary functions. The auxiliary functions are similar to the ones defined in Figure 4.9 except that they do not need to generate a feature expression from a configuration and a set of closed features.

4.4.3 VRA Denotational Semantics

Now that we have all required parts we define the denotational semantics of variational relational algebra using the denotational semantics of relational algebra.

Table accumulation function:

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

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 4.9: 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 $f\ x\ y = f(x, y)$.

We assume the existence of the function $rqSem : \underline{\textbf{Q}} \rightarrow \underline{\textbf{DBInst}} \rightarrow \underline{\textbf{Table}}$, which given a plain query and a plain database, returns a plain table named *result* according to the standard semantics of plain relational algebra. We then define the VRA denotational semantics $vqSem : \textbf{Q} \rightarrow \textbf{DBInst} \rightarrow \textbf{Table}$ as the accumulation of relational tables resulting from the semantics of its configured queries over their corresponding configured databases for all valid configurations of a variational database. The *mapRQSem* function 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

VRA denotational semantics:

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

Auxiliary functions for VRA denotational semantics:

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

Figure 4.10: Denotational semantics of variational relational algebra. Note that the query q is well-typed and explicitly annotated by the schema of the VDB instance $\mathcal{I}$.

database.[4] Finally, the *qToConfRelQs* takes a well-typed, explicitly annotated variational query and the set of valid configurations and configures the variational query for the given configurations and returns the set of configured queries paired with their corresponding configuration.

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

4.5 Variation-Minimization Rules

VRA is flexible, since an information need can be represented via multiple variational 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 variational queries they write by having different levels of variation. For example, consider the explicitly annotated query q_6 in Section 4.3.

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

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 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, that is, applying rules left-to-right, reduces redundancy while lifting them up, that is, applying rules right-to-left, makes a query more understandable. When applied left-to-right, the rules are terminating, since the scope of variation monotonically decreases in size.

Choice Distributive Rules:

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

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.

4.6 Variational Relational Algebra Properties

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 the *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 $\forall c \in \textbf{Config}.\ \mathbb{I}[\![\mathcal{I}]\!]_c = \mathcal{I}_i \implies \mathbb{Q}[\![q]\!]_c = \underline{q}_i$.

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 [58] 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, that is, $\mathit{type}(q, S) = A^e$ is $S \vdash q : A^e$, and $\underline{\mathit{type}}(\underline{q}, \underline{S})$ indicates RA's type system for the relational query $\underline{q}$ over the relational database schema $\underline{S}$, that is, $\underline{S} \vdash \underline{q} : \underline{A}$. 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, that is, $\underline{\mathit{type}}(\mathbb{Q}[\![q]\!]_c, \mathbb{S}[\![S]\!]_c)$, must be the same as the configured variational type of the variational query q with configuration c, that is, $\mathbb{A}[\![\mathit{type}(q, S)]\!]_c$. *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.

$$\begin{array}{ccc} \lfloor q \rfloor_S & \xrightarrow{\mathit{type}} & A^e \\ {\scriptstyle \mathbb{Q}[\![.]\!]_c}\downarrow & & \downarrow{\scriptstyle \mathbb{A}[\![.]\!]_c} \\ \underline{q} & \xrightarrow{\underline{\mathit{type}}} & \underline{A} \end{array}$$

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

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

Proof. By structural induction. We proved this theorem in the Coq proof assistant [48]. □

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

Theorem 4.6.2 implies that for all valid configurations of a VDB, any variational query is correlated to a relational query and since RA is type safe, its queries are type safe. Thus, variational queries are type safe.

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 4.4.1. 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.

4.6.3 Variation-Preserving Property at the Semantics Level

Since VDB contains multiple database variants, it is important that running a variational query over a VDB does not lose the variation encoded in a variational query and the VDB, that is, VRA's semantics must also be variation-preserving. This is visualized in the diagram below, where the vertical arrows indicate corresponding

configuration functions, *sem* indicates VRA's denotational semantics and $\underline{sem}$ indicates RA's denotational semantics. Note that *sem* runs a variational query on a VDB and $\underline{sem}$ runs a relational query variant over its corresponding configured relational database variant. Simply put, the configured variational table resulting from running the variational query q over the VDB $\mathcal{I}$ and variational schema S with the configuration c, $\mathbb{T}[\![sem(q,\mathcal{I})]\!]_{(type(q,S),c)}$, must be the same as the relational table resulting from running the configured relational query with configuration c over its corresponding relational database, $\underline{sem}(\mathbb{Q}[\![q]\!]_c, \mathbb{I}[\![\mathcal{I}]\!]_c)$. Clearly the diagram commutes: taking either path of 1) obtaining the semantics of the variational query $\lfloor q \rfloor_S$ first and then configuring it or 2) configuring $\lfloor q \rfloor_S$ first and then getting its semantics results in the same plain relational table. The variation-preserving property enforces the maintenance of variants that a tuple belongs to through running a query at the semantics level. This property follows from the denotational semantics of VRA since we define VRA's denotational semantics in terms of RA's denotational semantics by attaching either configurations of variants or feature expressions that denote multiple configurations of variants to the relational queries.

$$\begin{array}{ccc} \lfloor q \rfloor_S & \xrightarrow{sem} & t \\ {\scriptstyle \mathbb{Q}[\![\cdot]\!]_c}\downarrow & & \downarrow{\scriptstyle \mathbb{T}[\![\cdot]\!]_{(s,c)}} \\ \underline{q} & \xrightarrow{\underline{sem}} & \underline{t} \end{array}$$

Chapter 5 Variational Database Use Cases

Thus far we introduced the variational database framework and variational queries. However, some natural questions are: "How feasible is the variational database framework?", "How would an expert 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 describe preliminary work aimed at answering these questions. Thus, the goal of this chapter is twofold: first, to describe how generating a VDB can be made automatic; second, to guide an expert through both generating a VDB from a variation scenario when it cannot be done automatically and writing variational queries that express an 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, this information and these encodings are not available for us to use in order to evaluate variational databases. 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 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 [68].

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

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

query sets potentially biased toward queries containing more variation points since the focus of the papers is on variational parts of the system. A complete query set, capturing *all* information needs for each scenario might contain more plain queries, that is, queries that perform the same way over all variants. However, we do not believe this bias is harmful for the role the case studies are intended to serve, namely, motivating and evaluating variational database systems. For this role, queries that contain variation are more useful than plain queries, and additional plain queries can likely be more easily generated if needed.

We distribute the variational 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.

We first focus on variation in databases over space, Section 5.1. Section 5.1.1 describes the variation scenario from Hall [37] 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. Finally, Section 5.1.4 describes how variational queries capture the information need adapted from feature interactions described by Hall [37]. We then switch focus to variation in databases over time, in Section 5.2. Again, Section 5.2.1 describes the evolution of an em-

ployee database as the variation scenario from Moon et al. [56] 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 dataset.[2] Finally, Section 5.2.4 describes the adapted and adjusted queries from Moon et al. [56]. At the end of this chapter, Section 5.3, we discuss the trade offs of using variational databases and attempt to answer the question: "Should variation be encoded explicitly in databases?".

We distribute the VDBs, SQL scripts for generating them, and queries of our use cases.[3] We distribute the VDBs in both MySQL and Postgres in two forms, one intended for use with our VDBMS tool, and one intended for more general-purpose research on variation in databases. We distribute the variational queries as simple `#ifdef`-annotated SQL files to promote their broad reuse in the design and evaluation of other systems for managing variational relational data.

5.1 Variation in Space: Email SPL Use Case

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

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

to obtain a particular variant.

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

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

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

The email SPL use case shows the use of VDB to encode the variational information needs of a database-backed SPL. We consider an email SPL that has been used in several previous SPL research projects (e.g. [8, 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 [37]. The email SPL has been used in several previous SPL research projects (e.g. [6, 3]). The variational email database is populated using the Enron email dataset, adapted to fit our variational schema [65]. 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 [37] as high-level specification of a SPL.

2. We use the Enron email dataset[4] 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.

[4] `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 [37]:

- *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 [37]. 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 [37] 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 [37], 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.[5] 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 [65]. 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-

[5] 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 which is described in

Section 5.1.2,and one without the embedded schemafor use with our VDBMS tool in which the variational schema is provided separately.[6] 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 [37].. 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

[6]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 [29, 41] 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 [37]. 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 [68], and dealing with such changes is a well-studied problem in the database community known as *database evolution* [62].

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. [56] and populated by a dataset that is widely used in databases research.[7]

5.2.1 Variation Scenario: An Evolving Employee Database

Moon et al. [56] 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

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

Table 5.3: Evolution of an employee database schema from Moon et al. [56].

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}.

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

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.[8]

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

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

to the new schema [62]. 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.[9]

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.[10] We have tested the employee VDB for the properties described in Section 3.3 and all of them hold.

[9] All the scripts are available at: `https://zenodo.org/record/4321921`.
[10] 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. [56] 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.[11] 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., $({a_1}^{e_1}, {a_2}^{e_2})^e$.

[11] 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.

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

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:

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

Note that even though the attributes *name*, *firstname*, and *lastname* are not present in all three of the variants corresponding to V_3–V_5, the VRA encoding permits omitting presence conditions that can be completely determined by the presence conditions of the corresponding relations or attributes in the variational schema. So, $empQ_3$ 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 variational 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: *Should variation be encoded explicitly in databases?*

Expressiveness of explicit variation. The use cases in this chapter 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.[12]

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

[12]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.

of variation that is not well supported (e.g. query-level variation in SPLs), or when combining multiple forms of variation in one database (e.g. during SPL evolution).

We expect that ill-supported forms of variation are common in industry and justify the expressiveness of explicit variation. We illustrated an example of this in Section 1.1.2.

Complexity of explicit variation. The generality of explicit variation comes at the cost of increased complexity. The complexity introduced by presence conditions and choices is similar to the complexity introduced by variation annotations in annotative approaches to SPL implementation [46]. There is widespread acknowledgment that unrestricted use of variation annotations, such as the C Preprocessor's `#ifdef`-notation [35], makes software difficult to understand [50] and is error prone [34]. 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 [52]. 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 [33], virtual separation of concerns [45], or view-based editing [76, 69]. 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 [46]. 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 [14]. 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 [46]. 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 [73]), 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 [73]). Although not presented here, we build directly on this body of work, especially work on variational typing [22, 23], 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)

We implement a prototype of the VDB and VRA frameworks as the *Variational Database Management System (VDBMS)*. VDBMS is implemented in Haskell. VDBMS sits on top of any plain relational DBMS, which will store the data in the form of variational tables, explained in Section 3.2. The presence condition of tuples is stored as an attribute called "presence condition". Note that the rest of the presence conditions are stored on the Haskell side of the system. The presence conditions stored in the database are encoded as strings, unlike the presence conditions on the Haskell side of VDBMS, which are represented using an algebraic data type. To support running VDBMS with multiple different plain relational DBMS backends, we provide a shared interface for communicating with the back-end DBMS and instantiate it for different database engines such as PostgreSQL and MySQL. An expert can extend VDBMS to another database engine by writing methods for connecting to and querying from the database.

6.1 VDBMS Architecture

Figure 6.1 shows the architecture of VDBMS. We assume a VDB and its variational schema are generated by an expert and are stored in a DBMS. A VDB can be *configured* to its plain relational database variants by providing the configuration

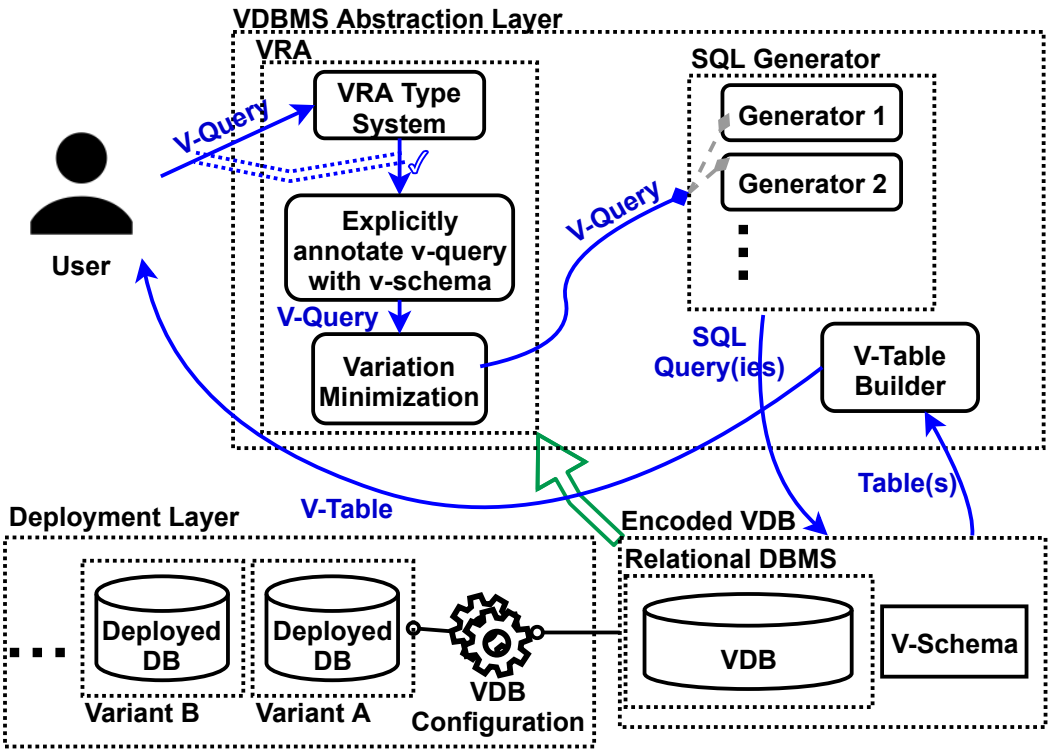


Figure 6.1: VDBMS architecture and execution flow of a variational query. The dotted double-line from the input variational query indicates the dependency of passing the variational query to this module only if it is valid. The dashed gray arrows with diamond heads demonstrate an option for the flow of the variational query. The blue filled arrows track the data flow, the green hollow arrows indicate an input to a module.

of the desired variant, Figure 3.2. For example, a SPL developer configures a VDB to produce software and its database for a client.

To extract information from a VDB, a user inputs a variational query q to VDBMS. First, q is checked by the *type system*. If the query is ill-typed, the user gets an error explaining what part of the query violated the variational schema. Otherwise, q is explicitly annotated by the schema and is passed to the *variation*

minimization module, to simplify q. The simplified query is then sent to the *generator* module where SQL queries are generated from variational queries by different approaches explained in Section 6.2.

All generated SQL queries are then executed on the underlying VDB. The result could be either a variational table or multiple variational tables, depending on the approach chosen by the SQL generator. The variational tables are passed to the *variational table builder* to create one variational table that filters out duplicate and invalid tuples, shrinks presence conditions, and eventually, returns the final variational table to the user. Note that the variational table builder module uses the accumulation functions introduced in Section 4.4.2 in addition to filtering out tuples and cleaning a variational table.

6.2 SQL Generators

Since VDBMS sits on top of a plain relational DBMS, in order to run variational queries we must translate them into (sets of) plain relational queries, which is the *SQL generator* module's task. These generators also add relation qualifiers to attributes in addition to adding presence conditions to projected attributes. We demonstrate this in Example 6.2.1 and Example 6.2.2. Then, VDBMS synthesizes the result into a variational table, which is the *variational table builder* module's task. Given an explicitly annotated, well-typed variational query q, we provide five approaches to generate SQL queries for q:

1. *Naive Brute Force (NBF)*: Configures a variational query q for all valid con-

figurations, that is, $\forall c \in \textbf{Config}$, translates them to RA queries, and finally generates SQL queries after renaming all subqueries and adding the projection of presence conditions to the set of projected attributes.[1] The SQL queries are sent to underlying DBMS and the results are gathered and cleaned up in the variational table builder module. Here is the flow of how results are generated by this approach:

$$q \xrightarrow{\mathbb{Q}[\![\mathrm{q}]\!]_c} [(c, \underline{q})] \xrightarrow[+\ \textit{project PC}]{\textit{generate SQL}} [(c, \underline{sql})] \xrightarrow{\textit{run queries}} [(c, \underline{table})] \xrightarrow{\textit{build table}} table$$

2. *Unique Brute Force (UBF)*: This approach is just like NBF except that we only generate SQL queries for unique RA queries produced by configuring the variational query. That is, it uses the unique variants function $\mathcal{Q}(q, F)$ introduced in Section 4.4.1. The implementation of this function is more efficient than its definition. That is:

$$q \xrightarrow{\mathcal{Q}(\mathrm{q,F})} [\underline{q}^e] \xrightarrow[+\ \textit{project PC}]{\textit{generate SQL}} [\underline{sql}^e] \xrightarrow{\textit{run queries}} [\underline{table}^e] \xrightarrow{\textit{build table}} table$$

Example 6.2.1. Consider query $q_1 = \pi_{empno^{V_4 \vee V_5}, name, firstname, lastname}(empbio)$ given in Example 4.1.1. Its corresponding SQL queries generated by either NBF or UBF that will eventually be run on the database. The feature expression representing groups of valid configurations are given below. For configurations $\{\}$ and $\{V_3\}$ (indicated by the feature expression

[1] Remember that variational queries written by a user does not explicitly project the presence condition attribute, however, to keep track of the variation associated with tuples.

$(\neg V_3 \wedge \neg V_4 \wedge \neg V_5) \vee (V_3 \wedge \neg V_4 \wedge \neg V_5))$ we have the query:

```
SELECT NULL
```

For configuration $\{V_4\}$ (indicated by the feature express $\neg V_3 \wedge V_4 \wedge \neg V_5$) we have the query:

```
SELECT empno,
       name,
       CONCAT( '(', t0.prescond, ')') AS prescond
FROM v_empbio AS t0
```

And finally, for configuration $\{V_5\}$ (indicated by the feature expression $\neg V_3 \wedge \neg V_4 \wedge V_5$) we have the query:

```
SELECT empno,
       firstname,
       lastname,
       CONCAT( '(', t0.prescond, ')') AS prescond
FROM v_empbio AS t0
```

Note that the only difference between NBF and UBF is that the former uses configurations and assigns them to SQL queries and then gathers their tables into a variational tables whereas the latter uses the feature expressions indicating a group of configuration and assigns them to SQL queries and then gathers their tables into a variational tables. For example, NBF uses the configurations given in this examples and runs the first SQL query twice whereas UBF uses the feature expressions and thus, runs each SQL only once. This does not have a big impact if the query is an empty query, as

in this example, but it could be significant in queries with more and larger shared variants.

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

$$q \xrightarrow{\mathcal{Q}(\mathrm{q,F})} [\underline{q}^e] \xrightarrow[+\ \textit{generate SQL}]{\textit{unify schema}} [\underline{sql}^e] \xrightarrow{\textit{inject PC}} [\underline{sql}] \xrightarrow{\textit{union}} \underline{sql}' \xrightarrow{\textit{run query}} \textit{table}$$

Example 6.2.2. Consider query $q_1 = \pi_{empno^{V_4 \vee V_5}, name, firstname, lastname}(empbio)$ introduced in Example 4.1.1. The final SQL query generated by the UAV approach is:

```
(SELECT empno,
        firstname,
        lastname,
        NULL AS name,
        CONCAT( '(', t0.prescond, ')', ') AND ('
              , '(NOT v3 AND v4) AND NOT v5'
              , ')') AS prescond,
 FROM empbio AS t0)
```

```
UNION ALL
 (SELECT empno,
         NULL AS firstname,
         NULL AS lastname,
         name,
         CONCAT( '(', t1.prescond, ')', ') AND ('
               , '(NOT v3 AND NOT v4) AND v5'
               , ')') AS prescond
  FROM empbio AS t1)
```

4. *Injected Naive Brute Force (NBF(i))*: Alternatively, the feature expression associated with a configuration can be injected inside the SQL query. This simplifies the variational table builder to just fix the schema of the returned tables. Example 6.2.3 illustrates this approach.

$$q \xrightarrow{\mathbb{Q}[\![q]\!]_c} [(c, \underline{q})] \xrightarrow[+\ \textit{inject PC}]{\textit{generate SQL}} [\underline{sql}] \xrightarrow{\textit{run queries}} [\underline{table}] \xrightarrow{\textit{build table}} \textit{table}$$

5. *Injected Unique Brute Force (UBF(i))*: Similarly, the injection of feature expressions into the SQL queries can be applied to the UBF approach. Example 6.2.3 illustrates this approach.

$$q \xrightarrow{\mathcal{Q}(q,F)} [\underline{q}^e] \xrightarrow[+\ \textit{inject PC}]{\textit{generate SQL}} [\underline{sql}] \xrightarrow{\textit{run queries}} [\underline{table}] \xrightarrow{\textit{build table}} \textit{table}$$

Example 6.2.3. Employing the NBF(i) or UBF(i) approaches to the query q_1 results in the following SQL queries. For configuration $\{V_4\}$ (indicated by the feature express $\neg V_3 \wedge V_4 \wedge \neg V_5$) we have the query:

```
SELECT empno,
       name,
       CONCAT( '(', t0.prescond, ')', ') AND ('
             , '(NOT v3 AND v4) AND NOT v5'
             , ')') AS prescond,
FROM v_empbio AS t0
```

For configuration $\{V_5\}$ (indicated by the feature expression $\neg V_3 \wedge \neg V_4 \wedge V_5$) we have the query:

```
SELECT empno,
       firstname,
       lastname,
       CONCAT( '(', t0.prescond, ')', ') AND ('
             , '(NOT v3 AND NOT v4) AND v5'
             , ')') AS prescond
FROM v_empbio AS t0
```

UAV attempts to reuse as much of the already existing results as possible. The generated SQL queries need to be independent from the underlying DBMS that stores the VDB. Hence, the SQL generator module has a submodule that renders generated SQL queries for each DBMS engine.

To ensure that these methods are implemented correctly we conducted two sanity checks:

1. We check that the variation-preserving property at the semantics level holds for all the methods, that is, we check that configuring the result produced by a method is the same as running the configured query over the corresponding configured database.

2. We check that the results from each pair of methods are equivalent based on the equivalency relation over variational sets defined in Section 2.4 (specifically, in Section 2.4).

Our set of queries for both the email VDB and employee VDB passed these sanity checks.

6.3 Experiments and Discussion

In this section, we compare the performance of VDBMS with regards to the approaches used to generate SQL queries introduced in Section 6.2. Note that the main contribution of this thesis is the new functionality that VDBMS adds to traditional RDBMSs. This contribution at its core provides vocabulary for developers and database administrators to explore and discuss variational needs. Additionally, it guarantees consistent encoding of variation in the database and queries as well as automatically checking the correct variational property in either the database or the queries. This results in removing the burden of manual manipulation of data and queries from developers and database administrators.

For our experiments, approaches introduced in Section 6.2 do not filter out tuples with unsatisfiable presence condition unless specifically mentioned otherwise. Accounting for filtering such tuples explicitly is indicated by adding *(f)* to the approach name. For example, NBF(f) is the Naive Brute Force approach that filters out tuples with unsatisfiable presence conditions. The variational queries

used for our experiments are available online[2] and they are described in Chapter 5. The runtime of queries contains all elements of an approach including the query passing the type system, being explicitly annotated by the variational schema, being optimized by the variation minimization rules, passed to the SQL generator, run on the VDB, and finally, generating the final variational table. We run the experiments on a MacBook Pro with a 2.4 GHz Core i7 processor and 8 GB of 1600 MHz DDR3 RAM. All experiments are run with PostgreSQL 13.3 as the database engine.

We conduct three main comparison of the approaches over our two use cases. The first one compares the five approaches with each other (Section 6.3.1). The second one investigates the effect of the number of (unique) variants on the approaches (Section 6.3.2). The last one investigates the effect filtering out tuples with unsatisfiable presence conditions (Section 6.3.3).

6.3.1 Analysis of Different SQL Generators

Figure 6.2 and Figure 6.3 show the runtime for each query for each of the five approaches introduced in Section 6.2 over the employee and email VDBs, respectively. The queries are labeled at the top of the plots while the approaches are indicated by different color bars. Figure 6.2 implies that NBF(i) mostly has a better runtime than NBF, however, the UBF and UBF(i) approaches do not follow such a trend for this dataset. Furthermore, NBF(i), UAV, and UBF have close

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

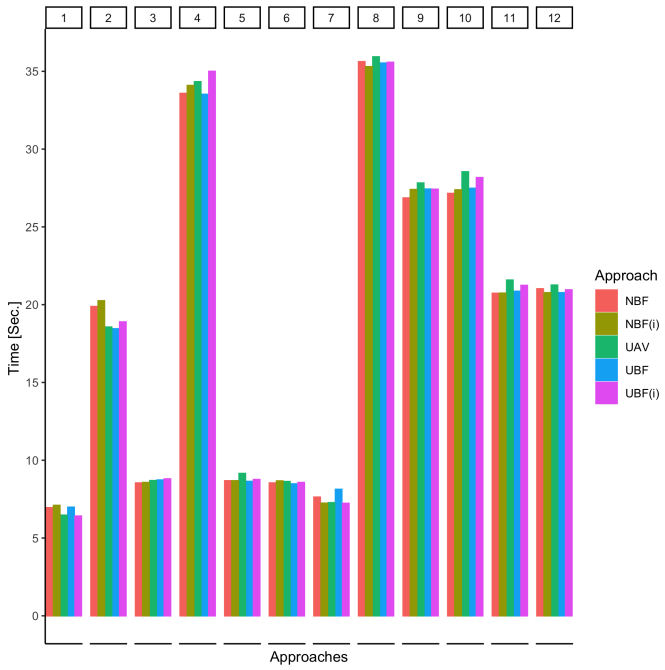


Figure 6.2: Comparison of SQL generators NBF, NBF(i), UAV, UBF, and UBF(i) over the employee VDB

performance to each other but none consistently performs better than the others.

Figure 6.3 also implies that NBF(i) consistently has a better runtime than NBF, and UBF(i) mostly has a better runtime than UBF for this dataset. While UAV mostly performs better than NBF(i), it is mainly comparable to UBF(i) for this dataset. Yet, UAV sometimes generates a non-runnable SQL query, showed by the striped bars in Figure 6.3. Example 6.3.1 explains this shortcoming in detail.

Based on our experiments, the query construction (from type system to generating SQL queries) takes similar time between the approaches. Their main difference comes down to the gross runtime of queries on the VDB and building the variational table. UAV does not take any time to build a variational table since the

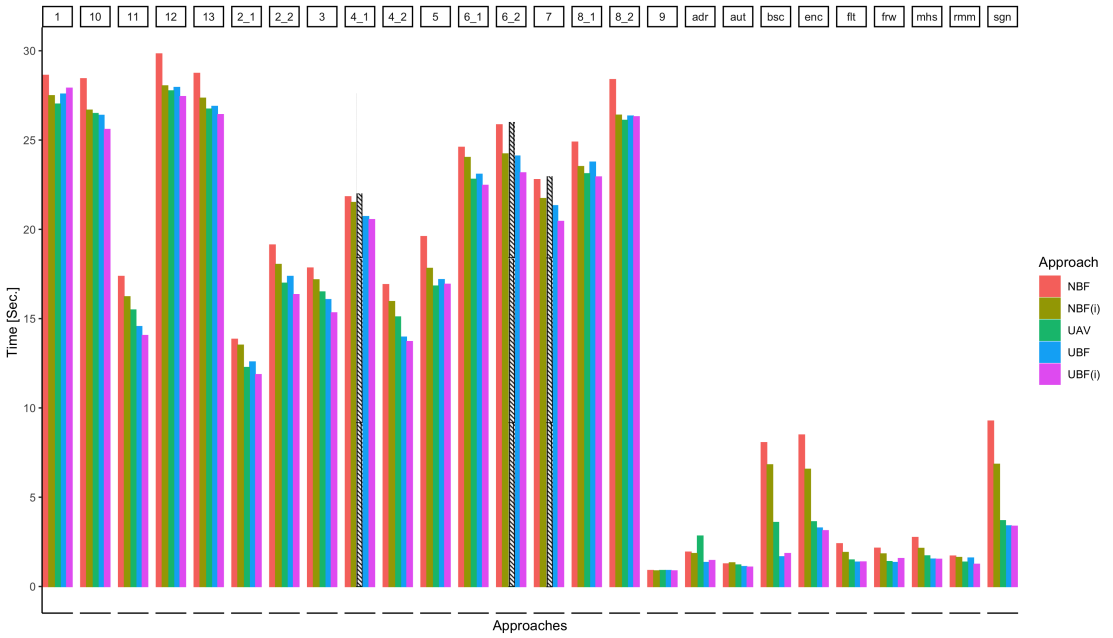


Figure 6.3: Comparison of SQL generators NBF, NBF(i), UAV, UBF, and UBF(i) over the email VDB

result already has the desired schema and presence conditions, however, it spends more time on running the SQL query since queries generated by UAV are usually more complicated. On the other hand, although NBF and UBF run multiple SQL queries per variational query their generated SQL queries are simpler than the ones generated by UAV. However, as opposed to UAV they have to adjust the returned table for each SQL query and apply the correct presence condition to the tuples. Finally, the main difference between the performance of NBF and NBF(i) (and similarly, UBF and UBF(i)) is where they apply the correct presence condition to the tuples. While NBF(i) and UBF(i) pass this task to the underlying database engine (which seems to perform better) the NBF and UBF approaches do this task

in the Haskell layer. Note that all four of these approaches still have to fix the schema of the returned tables to the variational table schema of the variational query.

Example 6.3.1. In this example we explain why the SQL query generated by the UAV approach sometimes cannot be run. PostgreSQL forces the type of an attribute a that is projected as `NULL` (that is, `NULL as` a) to be a string. Thus, using the union operation between subqueries when a has a different type causes an error. Assume the following query is generated by the UAV approach:

```
(SELECT body,
        NULL AS is_encrypted
 FROM messages)
UNION ALL
(SELECT body,
        is_encrypted
 FROM messages)
```

PostgreSQL forces *is_encrypted* to have the type string while in the second subquery it assumes the boolean type for *is_encrypted* since that is its defined type in the database. This causes a conflict in the assumption that subqueries of a query that uses the union operation must have the same schema and their attributes must have the same type. This issue can potentially be addressed by forcing the first subquery of the union to have all attributes projected and limit the number of returned tuples to zero. This would force the type of attributes as they are in the schema and since it does not return any tuples it will not change the result.

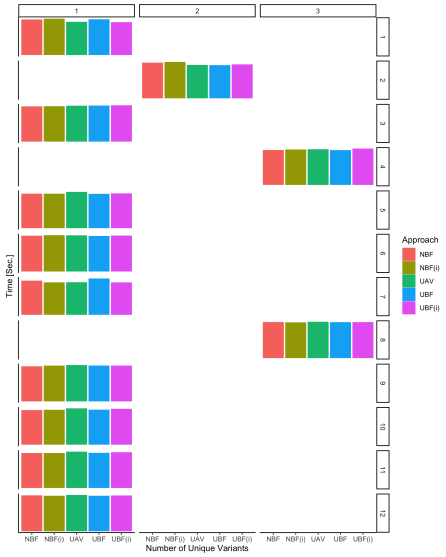
Section 6.3.2 sheds light on the impact of number of (unique) variants on each

approach and Section 6.3.3 explores the effect of the number of returned tuples on our approaches.

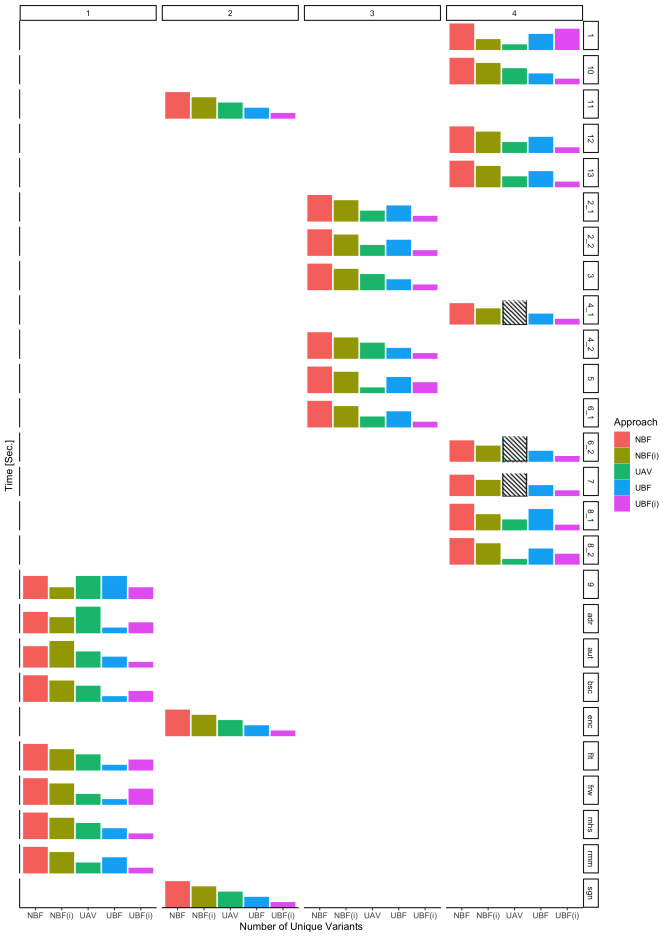
6.3.2 The Effect of Number of Variants on SQL Generators

In this section, we explore the effect of number of variants and uniquer variants of a variational query on its runtime in all approaches. Figure 6.4 illustrates the impact of the number of unique variants of variational queries in both the employee and email VDB. The categorical boxes on top and left side of the plots indicate the number of unique variants and the number/name of queries, respectively. Figure 6.4a does not provide much insight since the approaches all scale linearly with the number of unique. However, Figure 6.4b suggests an ordering of the approaches by their performance behavior, as $NBF \leq NBF(i) \leq UBF \leq UAV \leq UBF(i)$ for most queries of this dataset. Thus, the UAV and UBF(i) approaches seem to have a better performance as the number of uniquer variants increases.

We now explore the effect of the number of unique variants of queries compared to the number of variants. This difference would only affect the performance of two groups of approaches: 1) NBF vs. UBF and 2) NBF(i) vs. UBF(i). Remember that while NBF and NBF(i) naively process all variants of a variational query UBF and UBF(i) only process unique variants of a variational query. That is, the latter approaches do not unnecessarily repeat a query. This would not make much difference for empty queries, however, we hypothesize that it significantly affects the performance of NBF and NBF(i) for non-empty queries. We explore

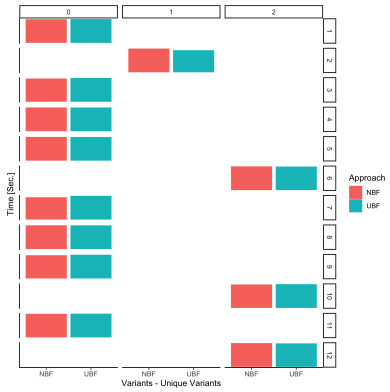


(a) The employee VDB.

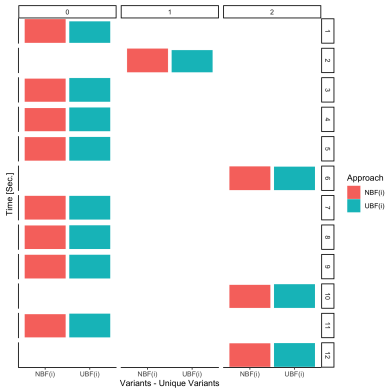


(b) The email VDB.

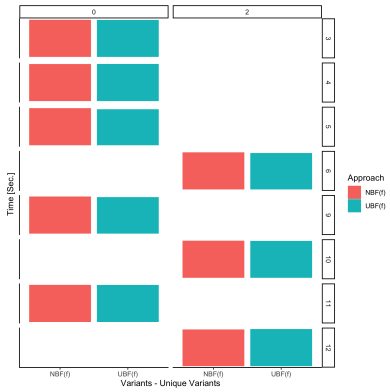
Figure 6.4: The performance of SQL generators as the number of unique variants of queries increases. The categorical boxes on the top of the plot show the number of unique variants for each query and the categorical boxes on the right demonstrate the query numbers/names. Note that the striped bars indicate a N/A value due to the limitation of the UAV approach explained in Section 6.3.1.



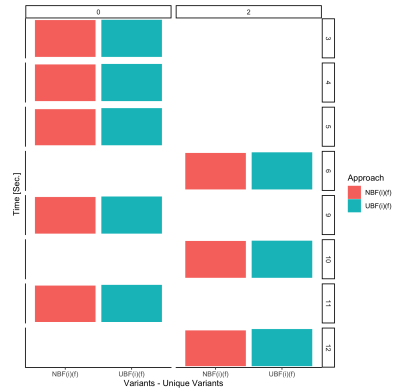
(a) Comparison of the NBF and UBF approaches.



(b) Comparison of the NBF(i) and UBF(i) approaches.



(c) Comparison of the NBF(f) and UBF(f) approaches.



(d) Comparison of the NBF(i)(f) and UBF(i)(f) approaches.

Figure 6.5: The effect of the difference of the number of variants and number of unique variants on the employee VDB. The categorical boxes on top and the left side of the plots illustrate the number of variants minus the number of unique variants and the number of queries, respectively.

this hypothesis by the plots shown in Figure 6.5 and Figure 6.6. In these plots, the categorical boxes on top and the left side of the plots indicate the number of variants minus the number of unique variants and the number/name of queries, respectively. Unfortunately, Figure 6.5 does not provide us with much insight. We

explore why this is the case in Section 6.3.3. However, Figure 6.6 confirms our hypothesis that as the number of differences between the number of variants and unique variants increases, the performance of NBF and NBF(i) decreases compared to UBF and UBF(i). This is due to repeatedly running an SQL query in addition to the cost of building the variational table for more relational tables. Notice that Figure 6.6 does not contain plots comparing approaches that filter out tuples with unsatisfiable presence conditions whereas Figure 6.5 does. We also explain the lack of these plots in Section 6.3.3.

6.3.3 The Effect of Filtering Invalid Tuples

In this section, we explore the effect of filtering out invalid tuples, that is, tuples with an unsatisfiable presence condition. Figure 6.7 and Figure 6.8 illustrate that filtering out invalid tuples increases the runtime of queries significantly. This increase is very significant for the UBF, UBF(i), and UAV approaches compared to NBF and NBF(i)[3] since the former approaches check the satisfiability of tuples' presence conditions while the latter applies the configuration to a tuple's presence condition and if it returns false the tuple is dropped. Thus, the calls to the SAT solver are more expensive than applying the configuration. Furthermore, Figure 6.7b, Figure 6.7d, Figure 6.8b, and Figure 6.8d illustrate that increasing the number of returned tuples reduces the performance of the NBF and NBF(i) approaches. A possible solution would be to either use an incremental SAT solver

[3]In our experiments these approaches took longer than 30 minutes for all queries from both datasets except for the ones showed in Figure 6.5c.

(a) Comparison of the NBF and UBF approaches.

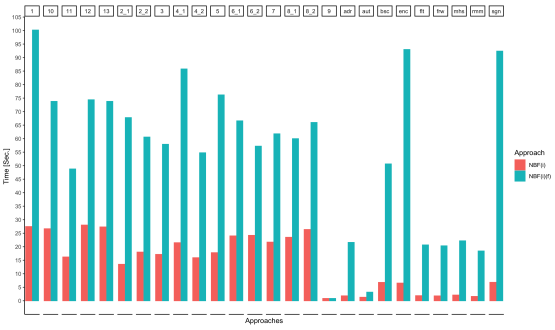
(b) Comparison of the NBF(i) and UBF(i) approaches.

Figure 6.6: The effect of the difference of the number of variants and number of unique variants on the email VDB. The categorical boxes on top and the left side of the plots indicate the number of variants minus the number of unique variants and the number/name of queries, respectively.

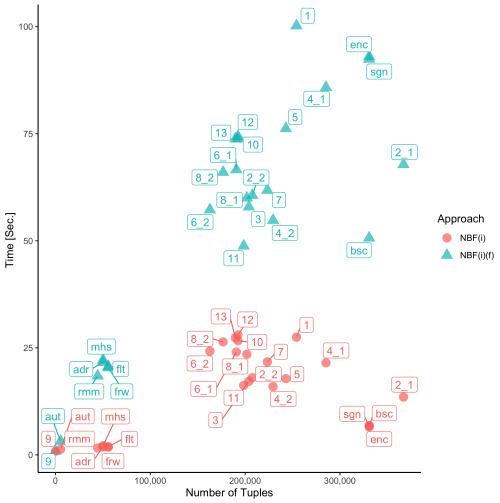
since most of the SAT problems have lots of common parts or to cluster the tuples based on their presence conditions and send unique presence conditions to the SAT

solver.

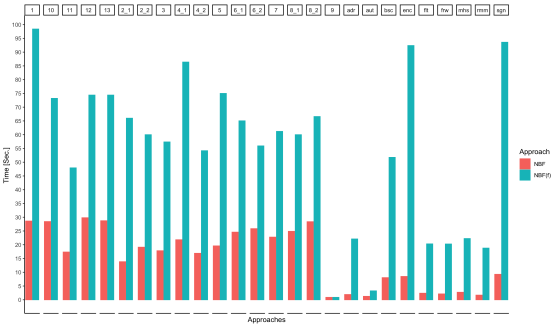
Finally, we explore the performance of approaches that filter out invalid tuples. As shown in Figure 6.9, there is not a clear ranking of the performance of the approaches that filter out invalid tuples and there is also not a clear connection as the number of tuples increases. This is due to the fact that the complexity of presence conditions also plays a role in the runtime of these approaches.



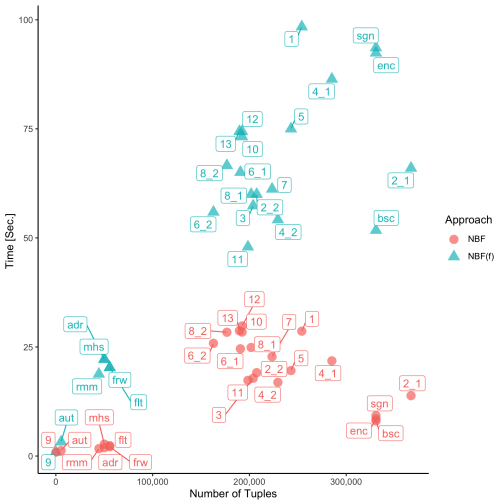
(a) The NBF(i) approach versus NBF(i)(f).



(b) The NBF(i) approach versus NBF(i)(f) as the number of returned tuples increases.

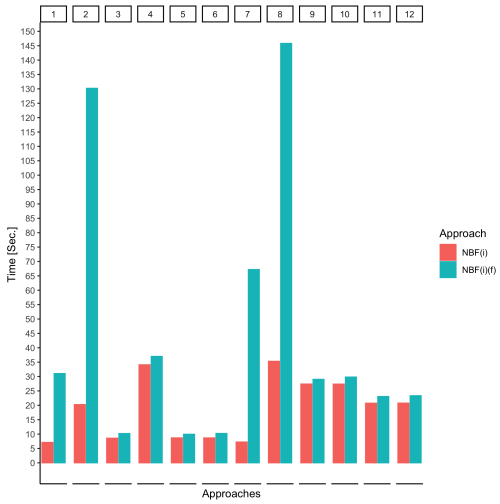


(c) The NBF approach versus NBF(f).

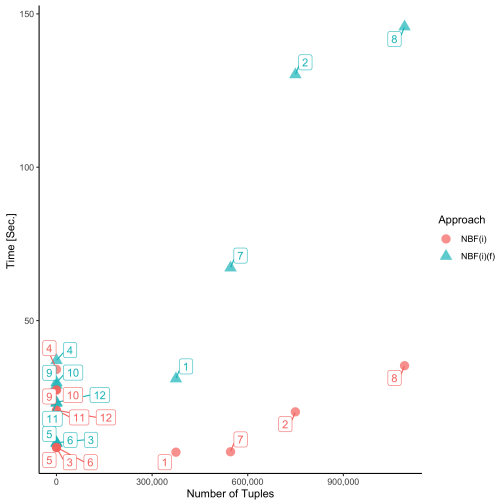


(d) The NBF approach versus NBF(f) as the number of returned tuples increases.

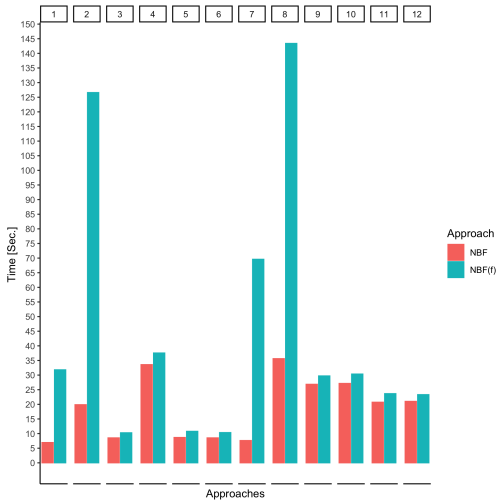
Figure 6.7: The effect of filtering out tuples with unsatisfiable presence conditions on SQL generator approaches over the email VDB.



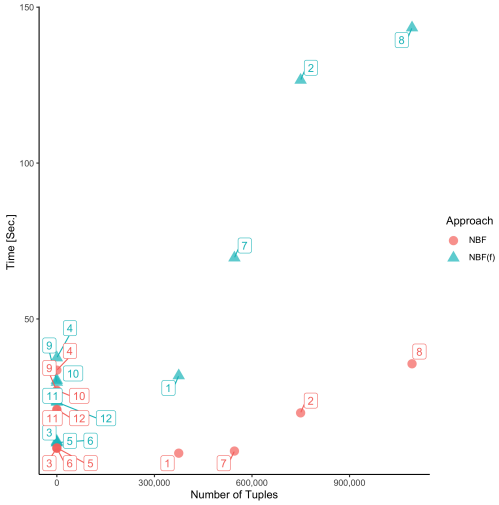
(a) The NBF(i) approach versus NBF(i)(f).



(b) The NBF(i) approach versus NBF(i)(f) as the number of returned tuples increases.

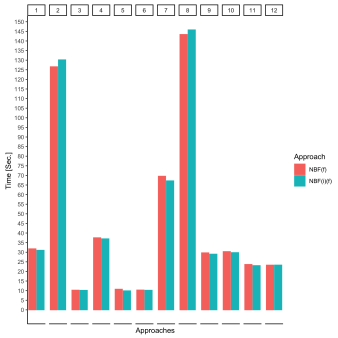


(c) The NBF approach versus NBF(f).

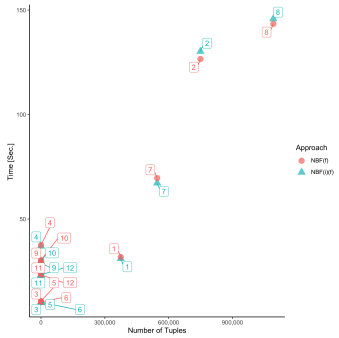


(d) The NBF approach versus NBF(f) as the number of returned tuples increases.

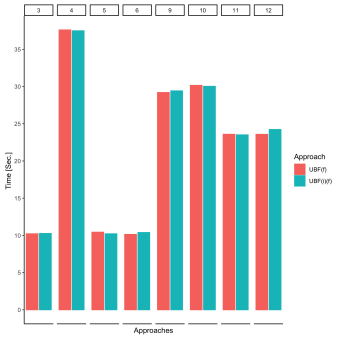
Figure 6.8: The effect of filtering out tuples with unsatisfiable presence conditions on SQL generator approaches over the employee VDB.



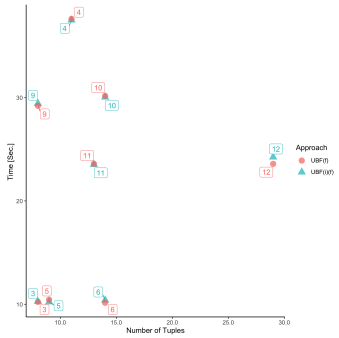
(a) Comparison of the NBF(f) and NBF(i)(f) approaches on the employee VDB.



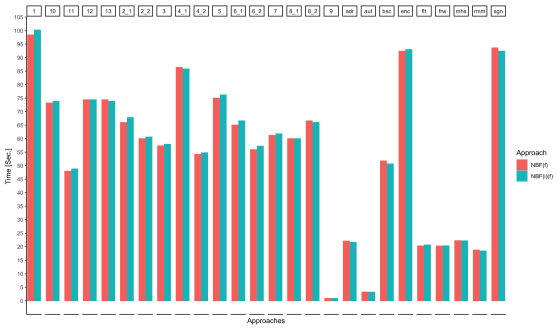
(b) Comparison of the NBF(f) and NBF(i)(f) approaches on the employee VDB.



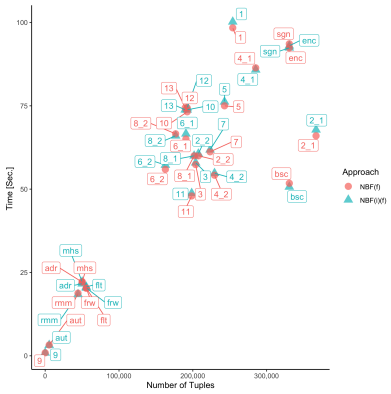
(c) Comparison of the UBF(f) and UBF(i)(f) approaches on the employee VDB



(d) Comparison of the UBF(f) and UBF(i)(f) approaches on the employee VDB



(e) Comparison of the NBF(f) and NBF(i)(f) approaches on the email VDB



(f) Comparison of the NBF(f) and NBF(i)(f) approaches on the email VDB

Figure 6.9: The effect of filtering out tuples with unsatisfiable presence conditions when presence conditions are injected into a query in on SQL generator approaches.

Chapter 7 Related Work

We have succeeded in providing a variational database framework and system by introducing a variation space for a database and explicitly accounting for variation in the database and the query language while relying on traditional non-variational RDBMSs. To the best of our knowledge, VDB and VDBMS are the first database framework and system that provide a generic solution to the problem of variation in databases. However, lots of work has proposed frameworks and tools to address specific kinds of variation. In this chapter, we study some of these tools and situate the variational database framework in the larger research context. In Section 7.1, we discuss multiple tools that address different kinds of variation in databases. In Section 7.2, we discuss tools that address the rise of variation in databases as a direct consequent of software development. Finally, in Section 7.3, we discuss some of the work on variational research.

7.1 Instances of Variation in Databases

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* [26, 56, 39, 61], 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* [18, 40] 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 [28]. 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 definition of variation is very limited in these problems. Such limitation allows for an efficient intelligent solutions, however, it tailors their solutions to a specific context and prevents one from using the same solution/system in a similar context when variation in time or space appears in a database [62]. For example, one cannot use a data integration system to manage variation in a database used in software produced by a SPL.

7.1.1 Schema Evolution

Current solutions addressing schema evolution rely on temporal nature of schema evolution. They use timestamps as a means to keep track of historical changes either in an external document [56] or as versions attached to databases [53, 19, 9, 67], i.e., either approaches fail to incorporate the timestamps into the database. Then, they take one of these approaches: 1) they require the DBA to design a

unified schema, map all schema variants to the unified one, migrate the database variants to the unified schema, and write queries only on the unified schema [39], 2) they require the DBA to specify the version for their query and then migrating all database variants to the queried version [53, 19, 9, 67], or 3) they require the user to specify the timestamps for their query and then reformulate the query for other database variants [56].

These approaches usually do not grant users access to old variants of data even if they desire so and it is messy to keep both different copies of a variant, one with the old schema and one with the unified schema, since every data addition/update now requires to be applied to all copies of the database variant. A better solution is maintaining a history of the changes applied to the database and the unified schema as an XML document and providing a language that allows users/developers to choose the variant they desire [56]. Unfortunately, this is achieved by limiting the schema evolution to temporal changes, offering a beautifully tailored approach for temporal changes, however, resulting in a non-extensible approach for non-temporal changes.

Temporal evolution is tracked by requiring the database to always have a time-related attribute in tables. Thus, queries have to specify the time frame for which they are inquiring information [56]. Now the user can choose a wide enough time frame in their queries to access to their desired variant(s). Aside from the detailed mapping of time frames and variants, this approach requires a query to have one and only one information need, no matter how many variants it is aiming. That is, if a time frame includes assumingely two variants a user cannot write a query

that extract two separate information needs for each of them accumulatively in one query. Even worse, if this query does not conform to one of the variant's schema but it conforms to the other one, the query still fails since there is no systematic way to identify that the query is ill (does not conform to the schema) for one of the variants. These limitations and constraint are the result of ignoring that temporal changes to a database is a form of variability.

7.1.2 Database Integration

The need for *data integration* systems was raised by the invention of the Internet and the World Wide Web which require quick access to lots of data stored online as well as the ability to query all of them. These systems need to query disparate data sources which often have different formats (e.g., some are completely structured data while others are either semi-structured or unstructured) and have been developed independently of each other [28]. Thus, work on *data integration* can be viewed as managing database variation in space at the content, schema, and format level. Most of these systems fall somewhere on the spectrum of warehousing and virtual integration. In the warehousing model, data from each source are loaded and materialized in a physical database called a warehouse whereas in the virtual integration model, the data remain in each data source and are accessed as needed at query time. The VDB framework falls in the middle of the spectrum since all database variants (data sources) are stored in a physical database without materialization.

In database integration systems, a *mediated schema* is defined for the integrated data and each data source has a wrapper/extractor that adjusts the schema and data of the source to the mediated schema. This is done by using schema matching and mapping approaches [60, 27, 16]. Our variational database framework skips this step since the variational schema contains within itself the variational encoding of the variants which is similar to the role of wrapper/extractor in database integration systems.

Finally, queries are posed in terms of relations in the mediated schema. Then, the database integration system reformulates and optimizes the query to grab parts of the data requested by the query from the corresponding data sources since all data sources do not necessarily contain the requested information in the query. Similarly, our SQL generators reformulate a variational query to extract data from corresponding variants. However, variational queries allow one to select the database variants (or parts of it) that they want to query whereas queries in a database integration system do not provide such an option. Furthermore, unlike VDBs, database integration systems are not variation-preserving, that is, the result has no indication as to the belonging of part of data to a specific variant (data source).

7.1.3 Temporal Databases and Database Versioning

There is a rich body of work on temporal databases which consider both the data model and the query language [44, 59, 71]. These databases manage data

that are temporal in nature, that is, the state of the data at a specific time is important, such as financial and medical data or record-keeping applications. Some of these databases use traditional relational databases and extend them to meet their needs [74, 4, 63], while others adopt an in-database approach [47]. These databases contain attributes indicating the start and end time, indicating either the transaction time or the valid time of the data. Temporal query languages usually extend traditional query languages by adding timestamps and conditions over timestamps [25, 43, 54]. Temporal databases only have variation at their content level. Thus, their model is simpler than the VDB framework. And similar to VRA, their query languages can express the desired time frame for extracting data.

Temporal databases support a linear timeline for a database, however, the rise of collaborative data science has led to non-linear time-based changes in a database. This resulted in work on database versioning that aims to support curating and analyzing data collaboratively [17]. Inspired by software version control systems such as Git, Huang et al. [40] introduce OrpheusDB which stores metadata information about the version graph such as the version number, its parent(s), checkout time, commit time, comments, etc. Using this metadata the shared parts of the database can be broken down at different scales so that the database for each version can be recreated. For example, each version could have a table of its own or tuples could have a version attribute and so on. OrpheusDB supports both git-style version control commands and SQL-like queries. Its query language *VQL* can query the data as well as the data and their versions. VQL

supports a subset of the query language for versioning and provenance proposed by Chavan et al. [21]. Bhattacherjee et al. [18] studies the trade-off of storage and recreation cost for different compression and optimization methods used to recreate a database version. Similar to temporal databases, database versioning systems also only contain variation at the content level. However, their query languages can express git-style commands whereas VRA cannot.

7.2 Database Variation Resulted from Software Development

A database may contain variation due to business requirements of software and its evolution [68, 38]. While some of schema evolution and database migration can be used to address this kind of database variation, other workarounds have been proposed as well. The first is that a different relational database may be *specified and created per-variant*, according to the information needs of each variant [57]. This approach is labor-intensive and difficult to maintain since changes need to be propagated across variants manually. The second strategy is to define a single *global schema that applies to all variants* [15]. This strategy is more efficient to maintain compared to the previous approach but is still hard to maintain, especially in face of SPL evolution. Due to lack of separation of concerns and suboptimal traceability of requirements to database elements [66] it is also complex, hard to understand, and unscalable [64]. Additionally, it suffers from design limitation and error-proneness since parts of the schema will be irrelevant to each variant, resulting in losing database's integrity constraints [64]. The third strategy is to

define a *variable data model* [66, 64, 2] which models a database schema (usually as an Entity-Relation model) with annotations of features from SPL to indicate their variable existence. Section 7.2.1 explains these approaches in more details.

7.2.1 Data Model Variability in Software Product Line

In this section, we focus on managing database variation in software product lines. The SPL community has a tradition of developing and distributing use cases to support research on software variation. For example, SPL2go [72] catalogs the source code and variability models of a large number of SPLs. Additionally, specific projects, such as Apel et al.'s [8] 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. Our use cases are the first resource that provide such datasets.

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 [49] 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. [42] 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. [64] generate a variable database schema from a given global schema and software configurations by mapping schema element to features. Siegmund et al. [66] 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. [42] is most similar to our notion of a variational schema since it is an annotative approach [46] 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 [49] is a compositional approach [46] 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 *variational queries*.

While the previous approaches all address data variation in space, Herrmann et al. [38] 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. They present the DAVE toolkit to address database evolution in SPL. Their approach generates a global evolution script from the local

evolution scripts by grouping them into a single database operations and executing them sequentially. This approach requires having the old and new schema of a variant to generate the delta scripts. However, it uses these scripts to ensure correct evolution of both data and schema at the deployment step.

7.3 Variational Research

In this section, we discuss some of the related variational research and other applications of it. The representation of variational schemas and variational tables is based on previous work on variational sets [31], which is part of a larger effort toward developing safe and efficient variational data structures [77, 55]. The representation of variation in variational queries is based on formula choice calculus [75, 41]. The central motivation of work on variational data structures is that many applications can benefit from maintaining and computing with variation at runtime [30, 24]. Implementing SPL analyses are an example of such an application, but there are many more [77].The ability to maintain and query several variants of a database at once extends the idea of computing with variation to relational databases.

VDBMS is not the only system that extends an existing system with variation. Grasley [36] expands on interpreters for variational imperative languages by providing a formal operational semantics for the variational imperative language VIMP. Alkubaish [5] investigates the use of algebraic effects to resolve the conflict between variation and side effects. Young et al. [78] add variation to SAT

solvers and argue that the variational SAT solver automates the interaction with the incremental solver.

Chapter 8 Conclusion

This thesis has presented the variational database framework as a generic solution to encoding multiple variants of a database collectively in one place for any kind of variation appearing in databases. The VDB framework is intended to systematically ensure that the variation in data and queries are encoded correctly and consistently. Thus, removing the burden of manual workarounds from database administrators and developers. If successful, it can improve the state of variation in databases research by providing a configurable database system for any kind of variation in databases. Thus, instead of developing a new database system each time a new kind of variation in databases arises, one can adjust and configure VDBMS to their need.

Section 8.1 briefly reviews the most important contributions of this thesis, and Section 8.2 provides some immediate directions for future work.

8.1 Summary of Contributions

The main contribution of this thesis is the variational database framework, a generic database framework that explicitly accounts for variation, and the variational database management system, a prototype of the VDB framework. We argued that variation is an orthogonal concern to databases and while there are

approaches that address it in specific contexts, there is no fundamental technique to address it in every contexts, especially in intersection of contexts. Hence, we incorporated variation as propositional formulas (Section 2.3) into the database while keeping track of variation while querying the database by employing approaches such as annotation, variational sets, and formula choice calculus, introduced in Section 2.4 and Section 2.5, respectively.

In essence, VDB systematically places multiple relational database variants in a single database while tracking their corresponding configurations at both the content and structure levels of a database (Chapter 3). This is a source of complexity that may impact understandability, as can be observed in our use cases introduced in Chapter 5. However, it also has several advantages: it is generic, in the sense that any set of database variants and queries can be encoded as a VDB and variational queries. Additionally, it enables direct association of variation in a database to variation in other parts interacting with the database such as software. We discussed that explicitly encoding variation in databases allows tracing variation between the program and data. It also empowers developers to check properties over a database to ensure that their desired constraints over a database hold (Section 3.3).

We also defined a variational query language (VRA) to extract information from VDBs (Section 4.1) along with its denotational semantics in terms of the relational algebra semantics (Section 4.4.3). VRA uses variational sets and formula choice calculus to incorporate variation into the relational algebra. Although explicitly accounting for variation in queries introduces more complexity in the language it

provides a systematic mechanism to ensure that the query follows the variation encoded in the database as well as possibly imposing new variation (Section 4.2). Still, this complexity is alleviated in two ways: 1) the queries are not enforced to repeat the variation encoded in the database, since queries can automatically be explicitly annotated by schemas (Section 4.3), and 2) the variation in queries are confluent, that is, the variation can move from one part of the query to another without changing its semantics due to syntactic equivalence rules (Section 4.5). Importantly, VRA is variation-preserving both at the type and semantics level (Section 4.6.2 and Section 4.6.3). That is, the corresponding variants of data is tracked throughout the execution of the query.

We provided two use cases that illustrate how software variation leads to corresponding variation in relational databases (Chapter 5). These use cases demonstrate the feasibility of VDBs and variational queries to capture the data needs of variational software systems. We argued 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 prototype. The case studies were developed by systematically combining existing data sources with software variation scenarios described in the literature. They each consist of a variational schema describing the structural variation of the database, the variational database itself containing the variational content, and a set of variational queries that satisfy realistic information needs over multiple variants of each database. These case studies can be used to 1) evaluate approaches and systems attempting to manage any kind of variation in databases, 2) learn how a variational database

can be generated from a scenario that describes such variation, and 3) design a system that automatically generates a variational database from non-variational databases and their corresponding variant. In particular, we used these use cases to evaluate VDBMS (Section 6.3).

Finally, we implemented the VDB framework and VRA query language as a variational database management system and compared different approaches of running a variational query on an underlying relational database engine (Chapter 6). Our experiments demonstrated that our different SQL generator approaches are comparable while filtering out tuples with unsatisfiable presence conditions takes a significantly long time, especially as the number of returned tuples grows.

8.2 Future Work

Since the long-term vision of this thesis is to support all kinds of variation in databases, we can imagine many applications of VDB and VDBMS. A measure of the success of this work will be if other researchers either use and configure VDBMS or pursue ideas in VDB to develop a new variation-aware database framework for different kinds of variation in databases, especially for new kinds of variation appearing in databases. However, in this section, we discuss a few immediate extensions and improvements to our VDB framework and its prototype that we plan on pursuing.

As described in Section 2.3, the VDB framework assumes a closed-world variation and configuration space, that is, the sets of features and configurations are

closed. An immediate improvement is having an open-world configuration space and then, an open-world feature space. This would be a great improvement since it makes adding new database variants and consequently, generating a VDB easier, resulting in a more accessible framework.

As described in Chapter 6, we implement presence conditions as strings, which is inefficient because we need to manipulate every tuple's presence condition while running a query and then, parsing it to a feature expression encoding to build the final returning variational table. We plan to implement presence conditions as a user-defined data type inside the underlying DBMS engine, which limits the underlying engines that VDBMS can rely on. Doing so would also require us to define user-defined functions over feature expressions inside the database engine to allow manipulation of them by VDBMS. Additionally, the performance of VDBMS can be improved by different optimizations, such as clustering returned tuples based on their presence condition to reduce the time it takes to filter out the tuples with unsatisfiable presence conditions.

Furthermore, to make VDBMS more accessible and configurable to each kind of variation in databases we can generalize parts of it. For instance, we plan to generalize the representation of feature expression such that its encoding and functions can be defined by an expert of the specific kind of variation. This would allow for context-specific optimizations with regards to the application domain of the kind of variation in databases.

Additionally, we plan to make VDBMS more useable by having a visual interface that shows a snapshot of the database as the user writes their query. This

would help the user understand the variation context of their query better since as the user writes their query they can see what parts of the database is accessible at the current variation context, instead of waiting until the query is completely written to see if it passes the type checker. This improvement requires a type system that allows for holes in queries. This can become richer by designing an error-tolerant type system that pinpoints where the user made a mistake in their query and allows the part of query that is well-typed to run. This is especially beneficial when a query has lots of encoded variation within it.

Finally, although we have not implemented a system to generate a VDB for a variational scenario it is trivial to do so if we have the variant databases. The problem is that, in most cases, the variant databases do not exist since current variational scenarios only simulate the effect of variation and do not incorporate it directly into the database or queries. Thus, an expert needs to manually generate the database variants. We plan to explore the possibilities of generating a VDB without having all database variants. This intertwines with moving from closed-world variation and configuration space to open-world. Another way to go about this would be to extend the query language such that not only it extracts data from a VDB but it also defines and manipulates data. The language can be further extended by allowing variation to be incorporated in the database constraints.

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] Mohammed Al-Kateb, Ahmad Ghazal, Alain Crolotte, Ramesh Bhashyam, Jaiprakash Chimanchode, and Sai Pavan Pakala. Temporal query processing in teradata. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 573–578, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450315975. doi: 10.1145/2452376.2452443. URL `https://doi.org/10.1145/2452376.2452443`.

[5] Ghadeer Alkubaish. Integrating Side Effects in Variational Programs Using Algebraic Effects. Master's thesis, Oregon State University, 2020. `https://ir.library.oregonstate.edu/downloads/0k225j37g`.

[6] 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.

[7] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer-Verlag, Berlin/Heidelberg, 2013. ISBN 978-3-642-37520-0.

[8] 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.

[9] 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.

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

[11] 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.

[12] Parisa Ataei, Fariba Khan, and Eric Walkingshaw. Managing variability in relational databases by vdbms, 2021. Draft.

[13] 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.

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

[15] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323?364, December 1986. ISSN 0360-0300. doi: 10.1145/27633.27634. URL https://doi.org/10.1145/27633.27634.

[16] Zohra Bellahsene, Angela Bonifati, and Erhard Rahm. *Schema Matching and Mapping*. Springer Publishing Company, Incorporated, 1st edition, 2011. ISBN 9783642165177.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Amit Chavan, Silu Huang, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. Towards a unified query language for provenance and versioning. *CoRR*, abs/1506.04815, 2015. URL http://arxiv.org/abs/1506.04815.

[22] 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.

[23] 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.

[24] 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.

[25] Jan Chomicki. Temporal query languages: a survey, 1995.

[26] 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`.

[27] Anhai Doan and Alon Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 26:83–94, 2005.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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`.

[33] 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.

[34] 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.

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

[36] Alex Grasley. Imperative Programming with Variational Effects. Master's thesis, Oregon State University, 2018. https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/z890s066h?locale=zh.

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

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

[39] 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.

[40] 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.

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

[42] 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.

[43] Christian S. Jensen and Richard T. Snodgrass. *Temporal Query Languages*, pages 3009–3012. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_407. URL https://doi.org/10.1007/978-0-387-39940-9_407.

[44] Christian S. Jensen and Richard Thomas Snodgrass. Temporal data management. *IEEE Trans. on Knowl. and Data Eng.*, 11(1):36–44, January 1999. ISSN 1041-4347. doi: 10.1109/69.755613. URL https://doi.org/10.1109/69.755613.

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

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

[47] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1173–1184. ACM, 2013. doi: 10.1145/2463676.2465293. URL https://doi.org/10.1145/2463676.2465293.

[48] Fariba Khan. Formal Verification of the Variational Database Management System. Master's thesis, Oregon State University, 2021. https://ir.library.oregonstate.edu/concern/graduate_thesis_or_dissertations/qf85nj87k?locale=en.

[49] 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.

[50] 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.

[51] Qiaoran Li. Application of the Variational Database Management System to Schema Evolution and Software Product Lines. Master's thesis, Oregon State University, 2019. https://ir.library.oregonstate.edu/concern/graduate_projects/hd76s6046.

[52] 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.

[53] 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`.

[54] L. Edwin McKenzie and Richard Thomas Snodgrass. Evaluation of relational algebras incorporating the time dimension in databases. *ACM Comput. Surv.*, 23(4):501–543, December 1991. ISSN 0360-0300. doi: 10.1145/125137.125166. URL `https://doi.org/10.1145/125137.125166`.

[55] 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.

[56] 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`.

[57] Marco Mori and Anthony Cleve. Feature-based adaptation of database schemas. In Ricardo J. Machado, Rita Suzana P. Maciel, Julia Rubin, and Goetz Botterweck, editors, *Model-Based Methodologies for Pervasive and Embedded Software*, pages 85–105, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38209-3.

[58] 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`.

[59] G. Ozsoyoglu and R.T. Snodgrass. Temporal and real-time databases: a survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995. doi: 10.1109/69.404027.

[60] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, December 2001. ISSN 1066-8888. doi: http://dx.doi.org/10.1007/s007780100057. URL `http://dx.doi.org/10.1007/s007780100057`.

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

[62] 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`.

[63] Cynthia M. Saracco, Matthias Nicola, and Lenisha Gandhi. A matter of time: Temporal data management in db2 for z. 2010.

[64] 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.

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

[70] 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.

[71] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass. *Temporal Databases: Theory, Design, and Implementation*. Benjamin-Cummings Publishing Co., Inc., USA, 1993. ISBN 0805324135.

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

[73] 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.

[74] Kristian Torp, Christian S. Jensen, and Richard T. Snodgrass. Stratum approaches to temporal DBMS implementation. In Barry Eaglestone, Bipin C. Desai, and Jianhua Shao, editors, *Proceedings of the 1998 International Database Engineering and Applications Symposium, IDEAS 1998, Cardiff, Wales, UK, July 8-10, 1998*, pages 4–13. IEEE Computer Society, 1998. doi: 10.1109/IDEAS.1998.694346. URL `https://doi.org/10.1109/IDEAS.1998.694346`.

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

[76] 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.

[77] 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.

[78] Jeffrey M. Young, Eric Walkingshaw, and Thomas Thüm. Variational satisfiability solving. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, SPLC '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375696. doi: 10.1145/3382025.3414965. URL https://doi.org/10.1145/3382025.3414965.