AN ABSTRACT OF THE PROJECT OF

Qiaoran Li for the degree of Master of Science in Computer Science presented on Dec 9, 2019.

Title: Application of the Variational Database Management System to Schema Evolution and Software Product Lines

Abstract approved: _____

Eric Walkingshaw

As a general solution to the problem of managing structural and content variability in relational databases, in previous work we have introduced the Variational Database Management System (VDBMS). VDBMS consists of a representation of a variational database (VDB) and a corresponding typed query language (v-query). However, since this is a novel database representation, there are no existing instances of VDBs or v-queries that can be used to evaluate the VDBMS. In this project, we present two case studies to demonstrate the use of VDBMS and support its evaluation. The case studies were constructed by systematically encoding variability scenarios from prior work and generating corresponding VDBs by adapting existing widely-used data sets. The first case study shows how to use the VDBMS to manage database variants under a schema evolution scenario. The second case study demonstrates how to integrate the VDBMS with a database-backed software product line. Each case study provides a VDB and a set of v-queries that will be used to evaluate the VDBMS. Additionally, we provide some insights into generating VDBs from relational databases that could assist future VDBMS users.

Application of the Variational Database Management System to Schema Evolution and Software Product Lines

by

Qiaoran Li

A PROJECT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented Dec 9, 2019
Commencement June 2020

Master of Science project of Qiaoran Li presented on Dec 9, 2019.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

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

Qiaoran Li, Author

ACKNOWLEDGEMENTS

First and foremost, I would like to give a special thanks to my advisor, Eric Walkingshaw. Thank you for offering me the opportunity to be your MS student and work in the research group. Your kindness and support make my wonderful academic journey at Oregon State University. I will forever be indebted to all you have given me.

I would like to show my gratitude to my committee members: Martin Erwig and Arash Termehchy. Thank you for your excellent comments and feedback on my project.

Furthermore, many thanks go to my colleague Parisa Ataei, who has helped me a lot through the VDBMS case studies. Thanks for all the wonderful discussions we had and the constructive feedback you gave on my project. Besides, I would like to thank all members of the Lambda group for all the precious moments we have shared together during lectures and group meetings.

Last but not least, I would like to express the deepest appreciation to my parents for your love and support. You have helped me to follow my dreams, even though that meant to live so far apart from each other. Also, I would like to thank my extended family, thank you all for your care and encouragement. I am so grateful for having such an incredible family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

Variability within a database creates numerous database variants [2]. For example, numbers of variants of schemas and data contents will be introduced in a database under schema evolution, and variability in software product lines (SPLs) requires various databases to support families of software products. While there are solutions to handle the database variation in each example, there is no general approach to manage the variability within the database. To deal with the database variants under schema evolution, one existing solution requires the database administrators (DBAs) to propagate schema changes into a unified schema, and then convert the existing data into the unified schema [13, 6]. And the common approach to manage database variations in database-backed SPLs is to create a global database [7] which contains the relations and attributes of all database variants. These approaches are inefficient and error prone, because some relations and attributes are not valid for some of the database variants, producing a lot of null values exposed to the user. Also, the user has to write multiple queries to express the same intent over different database variants.

A system, called the Variational Database Management System (VDBMS), provides a framework to manage variability in a database to meet the requirement of different contexts. It offers a variational database (VDB), which allows the user to incorporate the variability in a database, and a typed variational query language (v-query) that enables the user to request data from different variants [2]. The VDBMS has already been implemented, however, there is no existing instance of variational database and v-query that can be used to evaluate the VDBMS. Case studies are needed to be presented to use the VDBMS in real-world variation challenges and generate their counterpart VDBs and a set of v-queries for the experiment.

In this project, we present two case studies where we investigate the use of VDBMS in schema evolution and software product lines. In each case study, we systematically adopt an existing example with a widely used data set. The resulting VDB and v-queries for each case study will be used to evaluate the VDBMS. Additionally, the demonstration of generating VDB and v-queries can assist future VDBMS users to apply the VDBMS

in their applications.

The project was conducted with two goals as follows:

- Investigate the use of VDBMS to solve the schema evolution problem in Chapter 3.
- Investigate the use of VDBMS to fix the potential feature interactions and support different configurations in an SPL in Chapter 4.

The rest of this paper is organized as follows: Chapter 2 presents the basic concept of VDBMS with definition of some keywords. In order to show how to apply the VDBMS to schema evolution and SPLs, two case studies are present in Chapter 3 and Chapter 4. In Chapter 5, we concludes the paper with the future work.

Chapter 2 Background

VDBMS provides an approach to represent general forms of database variants in a compact, expressive, and structural way [4]. It incorporates variability into relational database elements including schemas, relations, attributes, and table contents. Additionally, VDBMS provides the user a query language, *variational query*, to inquire multiple database variants simultaneously and a strong type system to ensure the query conformed to the schema and its variability [2].

The following sections provide an overview of the key concepts of variational databases and variational queries.

2.1 Variational Databases

In VDB, variability is organized as a set of features, where each feature can either be enabled (`true`) or disabled (`false`). A *feature expression* is a propositional formula constructed by features, which describes the condition when one or more variants are valid. The syntax of feature expressions is shown in Figure 2.1. For example, suppose a college database started with a schema S_1. Due to a new registration system introduced, the attribute *name* in S_1 is decoupled into two attributes *firstname* and *lastname*, yielding a new schema S_2. But the college still needs to use the schema S_1 to support their old registration system. The schemas S_1 and S_2 are shown as below:

$$S_1 = \{student(id, name)\}$$
$$S_2 = \{student(id, firstname, lastname)\}$$

As you can see, the attribute *id* is present in both S_1 and S_2, and the attribute *name* is changed to attributes *firstname* and *lastname* when S_1 evolves to S_2. To identify two versions of schema S_1 and S_2, we could use features V_1 and V_2, respectively. The feature expression for attribute *name* is $(V_1 \wedge (\neg V_2))$, since *name* exists in S_1 but not in S_2. Similarly, the feature expression for attributes *firstname* and *lastname* is $((\neg V_1) \wedge V_2)$.

Feature Expression Syntax:

$f \in \mathbf{F}$	::=	(*any feature name*)	*Feature Name*
$b \in \mathbf{B}$	::=	`true` | `false`	*Boolean Value*
$e \in \mathbf{E}$	::=	$b \mid f \mid \neg f \mid e \wedge e \mid e \vee e$	*Feature Expression*
$c \in \mathbf{C}$	=	$\mathbf{F} \rightarrow \mathbf{B}$	*Configuration*

Figure 2.1: Feature Expression Syntax.

Moreover, the feature expression for attribute *id* is $(V_1 \vee V_2)$, since attribute *id* exists in both S_1 and S_2.

A *variational set (v-set)* $X = \{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}$ is a set of elements annotated by feature expressions [3, 9, 17]. It can generate several different plain sets by enabling or disabling the features, and an element appears in a variant of a set only if its corresponding feature expression evaluates to `true`. An annotated element x with feature expression e is denoted by x^e. For example, the v-set $\{a, b^{V_1}, c^{V_2}\}$ can generate four plain set variants: $\{a, b, c\}$ if V_1 and V_2 are both enabled, $\{a, c\}$ if V_1 is disabled and V_2 is enabled, $\{a, b\}$ if V_1 is enabled and V_2 is disabled, $\{a\}$ if V_1 and V_2 are both disabled. Note that we omit the feature expression if the element has a feature expression as `true`, e.g., $a^{\texttt{true}}$ is equivalent to a.

A variational set itself can be annotated with a feature expression. A variational set annotated by feature expression e is denoted by $X^e = \{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}^e$. In particular, the features expression e expresses a condition which determines if the elements in X are valid. We can represent this property by using $\{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}^e \equiv \{{x_1}^{e_1 \wedge e}, \ldots, {x_n}^{e_n \wedge e}\}$.

In order to incorporate variability into the database, VDB supports *annotating* relations, attributes, and tuples with feature expressions. The feature expression attached to an element is called its *presence condition*, which determines the condition under which the element is present in a database. To continue the above college database example, $id^{V_1 \vee V_2}$ is an annotated attribute with feature expression $V_1 \vee V_2$, the attribute *id* will present in the database if the presence condition $(V_1 \vee V_2)$ evaluates to `true`.

A *feature model* is a feature expression that specifies global relationships/restrictions among features. Now we consider the feature model of the above college database example, given feature V_1 and V_2, we could use the feature model $m = (V_1 \wedge \neg V_2) \vee (\neg V_1 \wedge V_2)$ to express that exactly one of V_1 or V_2 must be `true`.

A *variational schema (v-schema)* is a set of variational relation schema, where a *variational relation (v-relation) schema* is an annotated relation accompanied by a set of annotated attributes. Since the v-schema is an annotated v-set, it follows from the definition $\{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}^e \equiv \{{x_1}^{e_1 \wedge e}, \ldots, {x_n}^{e_n \wedge e}\}$ that the actual presence condition of an attribute is the conjunction of its feature expression with its relation's feature expression and feature model, and the actual presence condition of a relation is the conjunction of its feature expression with the feature model. Thus, a *hierarch of feature expressions* is introduced in v-schema, that is, the feature model is enforced on the presence condition of its relations, and the feature model with presence condition of its relations is enforced on all of the belonging attributes. For example, the S_v in the following is the corresponding v-schema for the college database,

$$S_v = \{student(id, name^{V_1}, firstname^{V_2}, lastname^{V_2})\}$$

it includes an annotated v-relation *student* and a set of annotated attributes with valid features V_1 and V_2. Because of the feature expression hierarch, the actual presence condition for relation *student* is its feature expression $(V_1 \vee V_2)$ enforced with the feature model m. For the sake of simplicity, we can drop the feature expression of *student* since $(V_1 \vee V_2)$ will be always satisfiable if the feature model m is satisfiable. Similarly, the feature expression of *id* is dropped, the feature expression of attribute *name* becomes V_1 instead of $(V_1 \wedge (\neg V_2))$, and the feature expression of attributes *firstname* and *lastname* becomes V_2.

To manage variability in database contents, i.e., tuples, VDB annotates tuples with a presence condition in the database called *variational tables (v-tables)*. In a v-table, the plain relation schema $s = r\,(a_1, \ldots, a_k)$ will be transformed to new schema: $s' = r\,(a_1, \ldots, a_k, pc)$, where pc is a presence condition which determines if the tuple is present in the database, and a_i represents the attribute which belongs to the relation r. Table 2.1 shows an instance of v-table for the college database. For example, the tuple `(001, "Bob Smith")` from S_1 will be populated into the v-schema S_v as `(001, "Bob Smith", NULL, NULL,` V_1`)` and the tuple `(003, "Sam", "Davis")` from S_2 will be populated into S_v as `(003, NULL, "Sam", "Davis",` V_2`)`. Note that the `NULL` value will not be looked up in a certain variant, because the presence condition of those attributes determines whether a value is valid or not.

Table 2.1: Instance of V-Table for College Database

id	**name**	**firstname**	**lastname**	**presCond**
001	Bob Smith	NULL	NULL	V_1
002	Lily Dumphy	NULL	NULL	V_1
003	NULL	Sam	Davis	V_2
004	NULL	Caity	Lee	V_2

Variational Relational Algebra Syntax:

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

$\circ$ denotes set operators: union or difference.
A denotes the variational attribute set.

Variational Conditions:

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

b denotes boolean tags: `true` or `false`.
$\underline{a}$ denotes the plain attribute.
$\bullet$ denotes the comparison operators.
k denotes the contant value.

Figure 2.2: Variational Relational Algebra Syntax.

2.2 Variational Queries

Variational Relational Algebra (VRA) is a query language designed for VDBMS, which is much like a structured query language, but with a major difference: it includes choices and tags [8] which will be used to manage the variations.

A choice between the alternative VRA expressions q_l and q_r is written $e\langle q_l, q_r\rangle$, and the condition of the choice is represented by the feature expression e. The choice $e\langle q_l, q_r\rangle$ represents a query equivalent to q_l when e evaluates to `true`, or q_r otherwise.

Figure 2.2 shows the syntax of VRA. A query written in VRA is called a *variational query (v-query)*. Compared to a traditional query, a v-query allows a choice between two queries; The selection, projection, and join operations in a v-query are adjusted to take variation into account. For instance, the v-query:

$$vq = \pi_{(name^{V_1}, lastname^{V_2}, firstname^{V_2})}(\sigma_{V_1\langle id=001, id=002\rangle} student)$$

projects column *name* if V_1 evaluates to `true`, or projects columns *lastname* and *firstname* if V_2 evaluates to `true`, and selects tuples from *student* where $id = 001$ if V_1 evaluates to `true`, or tuples where $id = 002$ if V_1 evaluates to `false`. Table 2.2 shows the results of running query vq upon the v-table instance shown in Table 2.1.

Table 2.2: Result of V-Query vq

Result when only V_1 is enabled:		
id	**name**	**presCond**
001	Bob Smith	V_1
002	Lily Dumphy	V_1

Result when only V_2 is enabled:			
id	**firstname**	**lastname**	**presCond**
003	Sam	Davis	V_2
004	Caity	Lee	V_2

Chapter 3 Applying VDBMS to Schema Evolution

Most database systems require changes to their schema over time, and the database administrators (DBAs) must decide how and when to convert the old database to a new one. This process is error prone due to unnecessary deletion and misleading preservation of data and its structure [14]. To avoid those problems and achieve perfect evolution, it is required to maintain the original information under each version of the schema. Meanwhile, the database system needs to support temporal queries over historical database variants [11].

In this chapter, we present the use of the VDBMS to address the challenge of schema evolution. The case study demonstrates the ability of the VDBMS to archive a historical database and query multiple snapshots of an evolving schema in a single VDB, where we systematically adapted an existing employee database into a schema evolution scenario and generated a VDB from a widely used employee data set.

3.1 Employee Database Schema Evolution

We adapt a schema evolution example systematically from [11, 16]. Table 3.1 outlines the schema of the example. It shows an employee database which has 5 versions of a schema (S_1 to S_5) evolving over time denoted by time stamps T_1 through T_5.

The first schema version S_1 initially has three tables *engineerpersonnel*, *otherpersonnel* and *job* during T_1. The first two tables maintain the basic information about engineers and the rest of the employees, respectively. The table *job* keeps the job title associated with its salary.

After some time, the DBA recognizes the requirement of managing employee information uniformly. In T_2, the DBA merges the tables *engineerpersonnel* and *otherpersonnel* into one table named *empacct*, yielding the schema version S_2.

As the company expands, the database layout changes into a new schema S_3 at time T_3. The need of department information prompts the DBA to create a new table *dept* to store the department name, department number, and its manager.

Table 3.1: Schema Evolution of Employee Database

T_i	**Schema Versions**	S_i	
T_1	*engineerpersonnel* (*empno*, *name*, *hiredate*, *title*, *deptname*) *otherpersonnel* (*empno*, *name*, *hiredate*, *title*, *deptname*) *job* (*title*, *salary*)	S_1	hiredate < 1988-01-01
T_2	*empacct* (*empno*, *name*, *hiredate*, *title*, *deptname*) *job* (*title*, *salary*)	S_2	hiredate < 1991-01-01
T_3	*empacct* (*empno*, *name*, *hiredate*, *title*, *deptno*) *job* (*title*, *salary*) *dept* (*deptname*, *deptno*, *managerno*)	S_3	hiredate < 1994-01-01
T_4	*empacct* (*empno*, *hiredate*, *title*, *deptno*) *job* (*title*, *salary*) *dept* (*deptname*, *deptno*, *managerno*) *empbio* (*empno*, *sex*, *birthdate*, *name*)	S_4	hiredate < 1997-01-01
T_5	*empacct* (*empno*, *hiredate*, *title*, *deptno*, *salary*) *dept* (*deptname*, *deptno*, *managerno*) *empbio* (*empno*, *sex*, *birthdate*, *firstname*, *lastname*)	S_5	hiredate < 2000-01-28

Within time T_4, more personal information about employees is introduced into the database. Considering privacy, the employee data are decoupled into two separate tables: *empacct* stores the business-related information, and *empbio* keeps the private information about employees. It results in the schema version S_4.

Finally, the company decides to adopt a new policy to motivate employees, that is, the salary is tied to individual performance instead of title. Due to this change, the attribute *salary* is moved to table *empacct*, and the table *job* is dropped. Also, to support operations which require the separation of first name and last name, the previous attribute *name* is decomposed into two attributes: *firstname* and *lastname*. These modifications are applied at time T_5, leading to a new schema S_5.

Over time T_1 to T_5, we have a total of 5 schemas, 15 relations, and 55 attributes across all versions of the database.

3.2 Employee Variational Databases

In this section, we present how we construct a v-schema from the historical employee database variants and generate the data sets for both plain schema and v-schema.

Table 3.2: Variational Schema for Employee Database

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

3.2.1 Schema Transformation

To represent variability in the employee database due to schema evolution, we set feature V_1 though V_5, to identify the schema variant S_1 to S_5 respectively. Given five variants of schema and the corresponding features, we 1) annotate its attributes with feature expressions, 2) combine the same annotated attributes into one by disjointing their feature expressions, and 3) simplify their feature expression based on the hierarchy of feature expressions. The same processes are applied to the relations. Take the attribute *sex* as an example, we first annotate attribute *sex* in variants S_4 and S_5 with feature expression V_4 and V_5 respectively, then we combine sex^{V_4} and sex^{V_5} into $sex^{V_4 \vee V_5}$, after that, we simplify the $sex^{V_4 \vee V_5}$ into *sex*, because the features expression of relation *empbio* has already restrict the presence condition of $(V_4 \vee V_5)$.

Table 3.2 represents the v-schema we build from the five variants of the schema. The resulting v-schema has 1 schema, 6 relations and 27 attributes in total, most of them are present conditionally in some variants.

To restrict that only one variant feature can be enabled at a given time, i.e., one

variant schema is valid at a given time, we construct the feature model below:

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

3.2.2 Populating the Employee Databases

To simulate the schema evolution scenario in the employee database, we use an existing and widely used employee data set,[1] partition the data into five parts, and populate them into five plain schemas shown in the Table 3.1, respectively.

The original employee data set contains the information about 240,124 individuals who are the permanent employees (whose end of hire date is 9999-01-01), we partition all employee information into 5 parts, D_1 to D_5, based on their hire date. Each part of data (D_i) represents the temporal data introduced during the time T_i, i.e, the employees hired during the time T_i are present in the D_i.

There are two options to populate the partition data, D_1 to D_5, into five plain schemas: 1) Populate five parts of data into five schema variants; 2) As shown in Figure 3.1, given a target schema variant S_i, we migrate partition data D_{i-1} from the schema variant S_{i-1} into S_i accordingly, and then insert D_i into S_i, where $1 < i \leq 5$. We call the former as a *non-migrated partition* and the latter as a *migrated partition.*

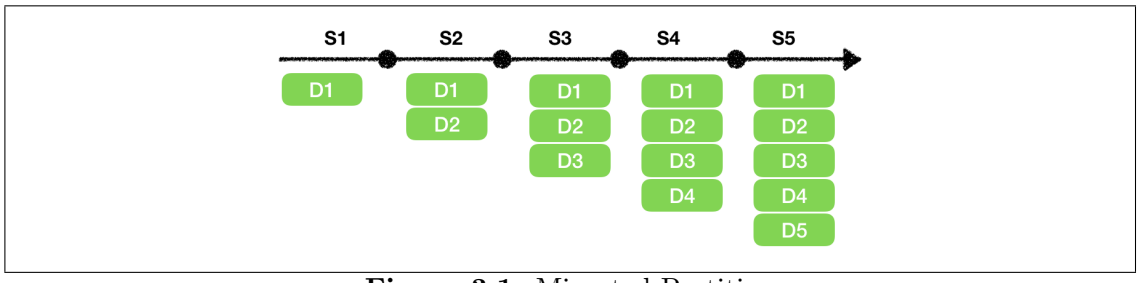

Figure 3.1: Migrated Partition

[1]The data is from `https://github.com/datacharmer/test_db` with some adaptation and edition.

The non-migration partition is more intuitive as all temporal data should exist in the corresponding temporal schema. However, it introduces situations that do not make sense. For instance, as shown in Table 3.1, suppose we have an engineer named Bob whose hire date is during time T_1, and he becomes a manager during time T_3. In this case, table *dept* in S_3 has a tuple with *managerno* as Bob's *empno*, but Bob's employee information is stored at *engineerpersonnel* in S_1, instead of at *empacct* in S_3. We therefore choose to use migrated partition for data sets in this case study, where every schema variant will contain data migrated from the previous variant.

For migrated partition, as shown in Table 3.1, we start populating the schema S_1 with employee data whose hire date is before 1988-01-01. We then populate the schema S_2 with employee data whose hire date is before 1991-01-01 (which contains the employee information in S_1 whose hire date is before 1988-01-01). The same strategy is applied to the rest of schema population.

After populating the data for each temporal schema variant, we map the data from the five schema variants into the v-schema accordingly. To perform the mapping, we map values of each attribute from the plain schema to v-schema and also insert its corresponding feature expression into attribute *presCond*. For example, for table *empacct* in schema variant S_2, we map the value from attributes *empno*, *name*, *hiredate*, *title*, and *deptname* into the corresponding attributes in *empacct* of v-schema, insert feature expression V_1 into VDB attribute *presCond*, and set value in other attributes that do not exists in S_2 to `NULL`. Note that the attribute *presCond* is not shown in Table 3.2 because it is a default attribute in VDB, and also the `NULL` value will never be looked up in a certain variant because the presence condition of those attributes determines whether a value is valid or not.

3.3 Examples of Variational Query

To give a sense of temporal queries over historical databases, we have written 14 v-queries for employee schema evolution. Each v-query represents a single intent which is potentially expressed differently in each database variant due to schema evolution. Note that the intents we use for this case study are taken from [11], and we adjusted the intents accordingly to fit into our context. For example, the intents in [11] use timestamps to identify their variants, we use features instead.

We have 2 kinds of intents: 1) The first kind of intent is to inquire the information from a single variant. To express this kind of intent, we only need one plain query q that requests the data from a specific variant, and then construct a v-query vq with the choice embedded. It expresses not only the intent that q represents but expresses the condition that q will be valid only if the choice evaluate to `true`. 2) The second kind of intent is to inquire the information from a group of variants. For this kind of intent, we write multiple plain queries to express the intent for different variants. To specify the condition that makes those plain queries work in their corresponding variants, we create a v-query vq to request data over multiple schema variants. We call the former *single variant intent* and the latter *multiple variant intent*.

In this report, we present queries in VRA's syntax in Figure 2.2. And we also encode the queries[2] presented in Section 3.3 and Section 4.4 into VRA's Haskell data type for evaluation use.

1. **Intent**: Return the salary value of the employee whose employee number (*empno*) is 10004 for VDB variant V_3.

   Classification: Single variant intent
   For variants V_3, the attributes *salary* and *title* exist in the relation *job*, and attributes *empno* and *title* exist in the relation *empacct*. To express the intent, we only need one query q to request data in schema variant V_3. The corresponding v-query vq not only represents the intent but expresses the condition that vq will be valid only if V_3 is `true`.

   $$q = \pi_{salary}(\sigma_{empno=10004}(empacct \bowtie_{empacct.title=job.title} job))$$
   $$vq = V_3\langle q, \varepsilon\rangle$$

2. **Intent**: Return the salary values of the employee whose employee number (*empno*) is 10004, for VDB variants V_3 to V_5.

   Classification: Multiple variant intent
   The attribute *salary* exists in the relation *empacct* when V_5 is `true`, or in the relation *job* when V_3, V_4, or V_5 evaluates to `true`. Thus we need 2 different plain

[2]The queries encoded in Haskell can be found in `https://github.com/lambda-land/VDBMS/wiki`

queries, q and q', to express the intent for this situation. To specify the condition the query works in its corresponding variants, we create a v-query vq to request data over three schema variants.

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

3. **Intent**: Return the manager's name (of department d001) for VDB variant V_3.

   Classification: Single variant intent
   To request the manager and department information on schema variant V_3, we construct query q and vq.

$$\begin{aligned} q =& \pi_{name}(\sigma_{deptno=d001}(empacct \bowtie_{empacct.empno=dept.managerno} dept)) \\ vq =& V_3\langle q, \varepsilon\rangle \end{aligned}$$

4. **Intent**: Return the manager's name (of department d001), for VDB variants V_3 to V_5.

   Classification: Multiple variant intent
   To request the information about manager's name for VDB variants V_3 to V_5, the attribute *name* exists in relation *empacct* when V_3 is `true`, and is contained in the relation *empbio* when V_4 is `true`. And the attributes *lastname* and *firstname* exist in *empbio* when V_5 is `true`. Additionaly, the relation *dept* in the variants V_3 to V_5 shares the same structure. Therefore, we need 3 different plain queries, q q', and q'', to express the intent in the different variants.

$$\begin{aligned} q =& \pi_{name}(\sigma_{dept.deptno=d001}(empacct \bowtie_{empno=managerno} dept)) \\ q' =& \pi_{name}(\sigma_{dept.deptno=d001}(empbio \bowtie_{empno=managerno} dept)) \\ q'' =& \pi_{(firstname,lastname)}(\sigma_{dept.deptno=d001}(empbio \bowtie_{empno=managerno} dept)) \\ vq =& V_3\langle q, V_4\langle q', V_5\langle q'', \varepsilon\rangle\rangle\rangle \end{aligned}$$

   The user can also express the information need in another way. Besides using

choices in queries, the user can also use a variational attribute set to express this intent. As shown in the following, we rewrite the query q' and q'' above into a new query q'''. Since the variants V_4 and V_5 share the same structure when we join the relations *empbio* and *dept*, we only need to annotate the attributes *name*, *firstname*, and *lastname* with corresponding feature expression.

$$q''' = \pi_{(name^{V_4}, firstname^{V_5}, lastname^{V_5})}(\sigma_{dept.deptno=d001}(empbio \bowtie_{empno=managerno} dept))$$
$$vq = V_3\langle q, (V_4 \vee V_5)\langle q''', \varepsilon\rangle\rangle$$

5. **Intent**: Find all managers that the employee 10004 worked with, for VDB variant V_3.

   Classification: Single variant intent
   To express this intent for variants V_3, we use q to join relation *empacct* with *dept* based on their *deptno*, and project the manager number from it.

$$q = \pi_{managerno}(\sigma_{empno=10004}(empacct \bowtie_{empacct.deptno=dept.deptno} dept))$$
$$vq = V_3\langle q, \varepsilon\rangle$$

6. **Intent**: Find all managers that employee 10004 worked with, for VDB variants V_3 to V_5.

   Classification: Multiple variant intent
   Based on v-schema presented in Table 3.2, the information required in the intent shares the same structure in the VDB variants V_3 to V_5, e.g, when V_3, V_4, and V_5 are `true`, the attributes *managerno* and *deptno* are all valid in the relation *dept*, and the attributes *empno* and *deptno* presents in the relation *empacct*. So we only construct one plain query q.

$$q = \pi_{managerno}(\sigma_{empno=10004}(empacct \bowtie_{empacct.deptno=dept.deptno} dept))$$
$$vq = (V_3 \vee V_4 \vee V_5)\langle q, \varepsilon\rangle$$

7. **Intent**: Find all salary values of managers, during the period of manager appointment, for VDB variant V_3.

Classification: Single variant intent

For variant V_3, to express the intent, we construct a plain query q to return the manager number with their salary.

$$\begin{aligned} q =& \pi_{(managerno, salary)}((dept \\ & \bowtie_{managerno=empno} empacct) \\ & \bowtie_{empacct.title=job.title} job) \\ vq =& V_3\langle q, \varepsilon\rangle \end{aligned}$$

8. **Intent**: Find all salary values of managers, during the period of manager appointment, for VDB variants V_3 to V_5.

   Classification: Multiple variant intent

   Based on the v-schema in Table 3.2 , the attribute *salary* exists in the relation *job* when V_3 or V_4 is `true`, and it exists in the relation *empacct* when V_5 is `true`. Thus, we need 2 different plain queries, q and q', to express the intent.

$$\begin{aligned} q =& \pi_{(managerno, salary)}((dept \\ & \bowtie_{managerno=empno} empacct) \\ & \bowtie_{empacct.title=job.title} job) \\ q' =& \pi_{(managerno, salary)}(empacct \bowtie_{empno=managerno} dept) \\ vq =& (V_3 \vee V_4)\langle q, V_5\langle q', \varepsilon\rangle\rangle \end{aligned}$$

9. **Intent**: Find the historical managers of the department where the employee 10004 worked for them in VDB variant V_3.

   Classification: Single variant intent

   For VDB variant V_3, we construct a plain query q, it first selects tuples where $empno = 10004$ from *empacct* by using *temp* and joins the result of *temp* with *dept*, then takes a final projection of *managerno*. Note that the VRA allows for renaming of attributes and queries similar to the relational algebra. For the

simplicity in this report, we use = to give a name to a sub-query.

$$
\begin{aligned}
temp =& \sigma_{empno=10004} empacct \\
q =& \pi_{managerno}(temp \bowtie_{temp.deptno=dept.deptno} dept) \\
vq =& V_3\langle q, \varepsilon\rangle
\end{aligned}
$$

10. **Intent**: Find the historical managers of department where the employee 10004 worked, in all history, for VDB variants V_3 to V_5.

    Classification: Multiple variant intent

    For this intent, the required data among variants V_3 to V_5 share the same structure, i.e, the attributes *managerno* and *deptno* are all valid in the relation *dept* when V_3, V_4, or V_5 evaluates to `true`, we construct one plain query q in this case.

$$
\begin{aligned}
temp =& \sigma_{empno=10004} empacct \\
q =& \pi_{managerno}(temp \bowtie_{temp.deptno=dept.deptno} dept) \\
vq =& (V_3 \vee V_4 \vee V_5)\langle q, \varepsilon\rangle
\end{aligned}
$$

11. **Intent**: For all managers that the employee, whose employee number (*empno*) is 10004, has worked with, find all the departments that the manager managed, for VDB variant V_3.

    Classification: Single variant intent

    To express the need, we construct a plain query q for variants V_3: we use *temp* to get the information about the manager that the employee 10004 has worked with, and then join the result from *temp* with *dept* again to select the department that manager managed. To make the query q only work when the schema variant is V_3, we create a v-query vq as following.

$$
\begin{aligned}
temp =& \pi_{(managerno,deptno)}(\sigma_{empno=10004}(empacct \bowtie_{empacct.deptno=empacct.deptno} dept)) \\
q =& \pi_{(dept.managerno,dept.deptno)}(temp \bowtie_{temp.managerno=dept.managerno} dept) \\
vq =& V_3\langle q, \varepsilon\rangle
\end{aligned}
$$

12. **Intent**: For all managers that the employee, whose employee number (*empno*) is

10004, has worked with, find all the departments that the manager managed, for VDB variants V_3 to V_5.

Classification: Multiple variant intent
The information needed for this intent has the same structure among VDB variants V_3 to V_5 based on v-schema Table 3.2, thus, we follow the same logic in intent 11 to construct a single plain query q.

$$\begin{aligned} temp =& \pi_{(managerno, deptno)}(\sigma_{empno=10004}(empacct \bowtie_{empacct.deptno=empacct.deptno} dept)) \\ q =& \pi_{(dept.managerno, dept.deptno)}(temp \bowtie_{temp.managerno=dept.managerno} dept) \\ vq =& (V_3 \vee V_4 \vee V_5)\langle q, \varepsilon \rangle \end{aligned}$$

13. **Intent**: For all managers, find all managers in the department that he/she worked in, for VDB variant V_3.

    Classification: Single variant intent
    To express the intent, we first get the manager number and department number from *dept* by using $temp_{,}$ and then join the *temp* with *dept* again to express the intent.

$$\begin{aligned} temp =& \pi_{(managerno, deptno)} dept \\ q =& \pi_{(deptmanagerno, deptname, dept.managerno)}(temp \bowtie_{temp.deptno=dept.deptno} dept) \\ vq =& V_3\langle q, \varepsilon \rangle \end{aligned}$$

14. **Intent**: For all managers, find all managers in the department that he/she worked in, for VDB variants V_3 to V_5.

    Classification: Multiple variant intent
    Since the related relation *dept* has no change during the schema evolution, we only need one plain query q to express the intent, which is explained in intent 13.

$$\begin{aligned} temp =& \pi_{(managerno, deptno)} dept \\ q =& \pi_{(dept.managerno, deptname, dept.managerno)}(temp \bowtie_{temp.deptno=dept.deptno} dept) \\ vq =& (V_3 \vee V_4 \vee V_5)\langle q, \varepsilon \rangle \end{aligned}$$

Chapter 4 Applying VDBMS to Software Product Lines

In this chapter, we present the use of VDBMS to handle variability in software product lines. The case study demonstrates the integration of the VDBMS to an email SPL, where we systematically adopt the prior work of a feature list and feature interactions in the email system and combine it with a real-world data set from the Enron Corp to generate an VDB for the email SPL and provide a set of v-query to request data across different software variants.

4.1 Software Product Line and Feature Interactions

A *software product line (SPL)* is a family of products sharing a set of reusable parts tailored to a specific requirement, the differences between those products are distinguished in terms of features [1]. An exponential number of software variants can be produced by enabling or disabling different features, which requires numerous database variants with different schemas and contents [4].

When several features are enabled, a *feature interaction* may occur. A feature interaction is an undesirable behavior of a software system due to the joint use of several features [5]. For instance, suppose we enable 2 features, auto-forward and auto-responder, in an email system. There is a user named Bob who sets auto-forward from his old address `old@ex.com` to a new address `new@ex.com`, and activates auto-responder for the address `new@ex.com`. There is an undesired interaction that occurs if a message is sent to `old@ex.com`, since it will be auto-forwarded to `new@ex.com`, and the auto-responder of `new@ex.com` will automatically generate a message back to `old@ex.com` and then forward back and forth, leading to a loop between these two email addresses.

Feature interactions are likely to happen in an SPL, since the products in an SPL share numerous features and each feature may be developed under different environments. When two features interact, the implementation of software products should change accordingly to prevent an unexpected error. As a part of implementation, the queries are required to be flexible to adapt to the changes.

These important requirements introduce a challenge to a database management system, namely managing the database variability in a software product line and providing explicit support for variations in queries. In practice, database-backed SPLs do not have a good way of dealing with variations, currently they use a global database [7] to include all relations and attributes needed for all variants of software products. This approach is error prone because it introduces numerous null values exposed directly to the user. To solve the problem in SPLs using VDBMS, we incorporate the variability of each product into a VDB and construct a set of v-queries to express the information need across several variants in an SPL.

4.2 Email SPL and Features

Several features in the Electronic mail (email) system have been developed since its original governing specifications were published [12]. We systematically adopt a relatively simple model of the email system and its feature list from [10]. In this email system, an individual writes a message, this message passes through one or more optional features until it reaches the client of recipients.

The email SPL consists of a basic functionality and 8 optional features, the optional features are shown as follow:

- ADDRESSBOOK: This feature allows the user to define a short alias for the recipient address.
- SIGNATURE[1]: This feature provides the user an ability to sign the message with a digital signature, and the recipient will verify the incoming signed message by using sender's verification key.
- ENCRYPTION[2]: This feature allows the user to encrypt the message by using the recipient's public key. When encrypted message delivered, the recipient can decrypt the message by using their own private key.

[1] We combined original features SIGNMESSAGE and VERIFYMESSAGE in [10] into SIGNATURE because the two features are synchronized.

[2] We combined original features ENCRYPTMESSAGE and DECRYPTMESSAGE [10] into ENCRYPTION because the two features are synchronized.

- AUTORESPONDER: This feature enables the user to set an auto-reply email message for incoming messages.
- FORWARDMESSAGES: This feature enables the user to forward every incoming message from an old email address to a new one automatically.
- REMAILMESSAGE: This feature allows the user to send anonymous messages. It requires the user to put the intended recipient in the first line of the email body and send it to the remailer server. The server will look up the provisioned pseudonym of the sender, replace the sender with a pseudonym, remove the first line of intended recipient, and send the modified email to the recipient.
- FILTERMESSAGES: This feature allows the user to filter the message by providing a list of address suffixes.
- MAILHOST: This feature is provisioned with a list of user-names in a mail host, it can check the email address of recipient against the list, if exists then hold the message until the recipient retrieves it, otherwise, the mail host sends back a non-delivery notification to the sender.

4.3 Email Variational Databases

In this section, we present how we construct a v-schema and VDB for the email SPL and populate the email VDB with data sets from five special product variants.

4.3.1 Schema Transformation

We base our case study on the Enron email corpus [15]. As shown in the Table 4.1, there are four relations (shown as black texts in Table 4.2) we adopt from the Enron schema: 1) *employeelist* stores the employee's information, in particular, the attribute *email_id* is the employee's email address, 2) *messages* stores message information, 3) *recipientinfo* stores the recipient's information where the attribute *rvalue* is the recipient's email address, and 4) *referenceinfo* stores information about the email forwarded or replied with the original email.

Table 4.1: Enron Schema

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

To make the Enron corpus support the optional features in the email SPL, we extend its schema with a set of relations and attributes (shown as orange texts in Table 4.2). We add relation: *auto_msg* to store the subject and body of the user's autoresponder email, *forward_msg* to map the user id to their email address that the user wants to forward messages to, *remail_msg* to map users to their provisioned pseudonyms, *filter_msg* to maintain the email suffix that the user sets to filter, *mailhost* to keep a list of user-names in a mail host, and *alias* to map the user's nickname to the email address. In addition, we add *public_key* and *verification_key* attributes in relation *employeelist* to store the user's public key and verification key, and also extend the relation *messages* with attributes *is_system_notification*, *is_signed*, *is_encrypted*, *is_autoresponse*, and *is_forward_msg* to indicate the email message's property. We call these *message property attributes*.

To make the schema of the email SPL variational, we identify the feature expression of relations and attributes, and tag the corresponding presence condition on them, yielding a v-schema for email SPL in Table 4.2. For each feature in the SPL, we defined a corresponding feature in VDB. To differentiate the features defined in SPL and that defined in VDB, we use the upper case for features in SPL and the lower case for features in VDB.

4.3.2 Populating the Email Databases

In this case study, we premise that the database variants of the email SPL contain the data from five products: Basic Email, Enhanced Email, Privacy-focus Email, Group email, and Premium email. Table 4.3 depicts the product map for the email SPL together with a set of enabled features for each product. All five email products support the core functionality of the email system. Basic Email has no optional features enabled. En-

Table 4.2: Variational Schema for Email Database

Variational Schema for Email SPL
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_signed*^{signature} , *is_autoresponse*^{autoresponder}, *is_forward_msg*^{forwardmessages}) *recipientinfo*(*rid*, *mid*, *rtype*, *rvalue*) *referenceinfo*(*rid*, *mid*, *reference*) *forward_msg*(*eid*, *forwardaddr*)^{forwardmessages} *filter_msg*(*eid*, *suffix*)^{filtermessages} *remail_msg*(*eid*, *pseudonym*)^{remailmessage} *auto_msg*(*eid*, *subject*, *body*)^{autoresponder} *alias*(*eid*, *email*, *nickname*)^{addressbook} *mailhost*(*eid*, *username*, *mailhost*)^{mailhost}

Table 4.3: Product Map for Email SPL

	Basic Email	**Enhanced Email**	**Privacy-focus Email**	**Groups Email**	**Premium Email**
Core Function	✓	✓	✓	✓	✓
ADDRESSBOOK				✓	✓
SIGNATURE			✓	✓	✓
ENCRYPTION			✓	✓	✓
AUTORESPONDER				✓	✓
FORWARDMESSAGES		✓			✓
REMAILMESSAGES			✓		✓
FILTERMESSAGES		✓			✓
MAILHOST				✓	✓

hanced Email supports the feature FORWARDMESSAGES and FILTERMESSAGES. Privacy-focus Email provides the user with features ENCRYPTION, SIGNATURE, and REMAILMESSAGE, it enables the user to send encrypted, signed, and anonymous messages. Group Email is designed towards the group usages such as business owners and organizations; it offers the user an ability to define a short alias for email address (ADDRESSBOOK), set autoresponder email (AUTORESPONDER), use the email hosting service (MAILHOST), and support the privacy property (ENCRYPTION and SIGNATURE). Premium Email enables all eight features. Based on feature configurations for those five email prodcuts, we build up the corresponding plain schemas for each product as shown in Table 4.4, where those plain schemas are five special configurations from the email SPL v-schema in Table 4.2.

We populate the Enron email dataset[3] to five plain schemas of software variants. The original Enron email dataset contains about 252,759 internal emails from 150 employees. We generate data for extended relations and attributes given in Section 4.3.1. For example, we create public key and private key as the *public_key* and *verification_key* for users who have ENCRYPTION and SIGNATURE enabled. Additionally, for each email message, we modify the data according to the client's capability of the sender and recipient. For example, an email message can only be encrypted if both sender and recipient support ENCRYPTION.

To fit the Enron email data sets to our context, we divide the 150 employees into 5 groups and consider them as users of the 5 email products respectively. To identify the product that an email message belongs to, we categorize the messages to Privacy-focus Email if both sender and recipient are the users of the Privacy-focus Email, while for other email messages we only check the sender's client and categorize them into their corresponding products. For example, if the sender uses Enhanced Email, no matter which email product the recipient uses, we simply make this email message sits in the database variant of the Enhanced Email.

Given 5 variants of the database for email SPL and a set of enabled features for each product, we populate the VDB of email SPL with the corresponding data annotated with their presence condition based on the enabled features. To tag the presence condition to tuples, we add an attribute *presCond* in each v-table, and insert the corresponding presence condition value in it. The presence condition of tuples is deter-

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

Table 4.4: Schema Variants for Email SPL

Product	Schema Variants
Basic Email	*employeelist*(*eid*, *firstname*, *lastname*, *email_id*, *folder*, *status*) *messages*(*mid*, *sender*, *date*, *message_id*, *subject*, *body*, *folder* , *is_system_notification*) *recipientinfo*(*rid*, *mid*, *rtype*, *rvalue*) *referenceinfo*(*rid*, *mid*, *reference*)
Enhanced Email	*employeelist*(*eid*, *firstname*, *lastname*, *email_id*, *folder*, *status*) *messages*(*mid*, *sender*, *date*, *message_id*, *subject*, *body*, *folder* , *is_system_notification*, *is_forward_msg*) *recipientinfo*(*rid*, *mid*, *rtype*, *rvalue*) *referenceinfo*(*rid*, *mid*, *reference*) *forward_msg*(*eid*, *forwardaddr*) *filter_msg*(*eid*, *suffix*)
Privacy-focus Email	*employeelist*(*eid*, *firstname*, *lastname*, *email_id*, *folder*, *status* , *verification_key*, *public_key*) *messages*(*mid*, *sender*, *date*, *message_id*, *subject*, *body*, *folder* , *is_system_notification*, *is_signed*, *is_encrypted*) *recipientinfo*(*rid*, *mid*, *rtype*, *rvalue*) *referenceinfo*(*rid*, *mid*, *reference*) *remail_msg*(*eid*, *pseudonym*)
Group Email	*employeelist*(*eid*, *firstname*, *lastname*, *email_id*, *folder*, *status* , *verification_key*, *public_key*) *messages*(*mid*, *sender*, *date*, *message_id*, *subject*, *body*, *folder* , *is_system_notification*, *is_autoresponse*, *is_signed*, *is_encrypted*) *recipientinfo*(*rid*, *mid*, *rtype*, *rvalue*) *referenceinfo*(*rid*, *mid*, *reference*) *auto_msg*(*eid*, *subject*, *body*) *alias*(*eid*, *email*, *nickname*) *mailhost*(*eid*, *username*, *mailhost*)
Premium Email	*employeelist*(*eid*, *firstname*, *lastname*, *email_id*, *folder*, *status* , *verification_key*, *public_key*) *messages*(*mid*, *sender*, *date*, *message_id*, *subject*, *body*, *folder* , *is_system_notification*, *is_encrypted*, *is_signed*, *is_autoresponse* , *is_forward_msg*) *recipientinfo*(*rid*, *mid*, *rtype*, *rvalue*) *referenceinfo*(*rid*, *mid*, *reference*) *forward_msg*(*eid*, *forwardaddr*) *filter_msg*(*eid*, *suffix*) *remail_msg*(*eid*, *pseudonym*) *auto_msg*(*eid*, *subject*, *body*) *alias*(*eid*, *email*, *nickname*) *mailhost*(*eid*, *username*, *mailhost*)

mined by which products it belongs to. For example, the presence condition for tuples from Privacy-focus Email is ($signature \wedge encryption \wedge remailmessage \wedge (\neg(addressbook \vee autoresponder \vee forwardmessages \vee remailmessage \vee filtermessages \vee mailhost))$), it represents the condition that there are only 3 features enabled (SIGNATURE, ENCRYPTION, REMAILMESSAGE), and the other 5 features are disabled.

For attributes in the v-schema that do not exist in a certain schema variant, we simply set NULL as the value for the corresponding attributes in the VDB. For instance, the attribute *verification_key* is not in the schema of Enhanced Email, we set `NULL` as the value of *verification_key* for tuples from Enhanced Email in the VDB. Recall that the `NULL` value will never be looked up in a certain variant because the presence condition of those attributes determines whether a value is valid or not.

4.4 Examples of Variational Query

In this section, we present v-queries[4] that express the intents across different software variants, we present a set of v-queries according to the functionality of the features and feature interactions in the email SPL.

The queries are written from a software developer's perspective to deal with the feature interactions. When an message is incoming to a email client, the software should generate different response message based on the feature configuration. As a part of implementation, the developer needs to inquire the feature-related data, and those data are only valid when the feature is enabled. When developers combine those features into one program, they need to change the code and the related query accordingly to fix the features interaction. In this section, we present a set of v-query examples that can express those information needs.

4.4.1 V-Queries for Features

We provide 8 v-queries to request the data that is used to generate an outgoing email message for a single feature functionality. For each feature, we use to represent a plain query which requests the feature-related data to generate the email message, and $vq_{feature}$ to represent a v-query which express the same intent with choices embedded.

[4]The queries encoded in Haskell can be found in `https://github.com/lambda-land/VDBMS/wiki`

1. **Intent**: Given a message X, return the recipient's nickname in feature ADDRESSBOOK.
   Query: To express the intent, we write query $q_{addressbook}$ to project a *nickname* with the recipient email address (captured by the attribute *rvalue*), it uses subquery q_{rec_eid} to get the recipient's *eid* and its email address (*rvalue*), and then join the result with *alias* to return the corresponding nickname. The corresponding v-query $vq_{addressbook}$ can also express the intent with a choice embedded, it not only expresses the intent but expresses the condition that $q_{addressbook}$ will be valid only if *addressbook* is `true`.

$$\begin{aligned} q_{rec\_eid} =& \pi_{(eid,rvalue,mid)}((\sigma_{mid=X}\, recipientinfo) \\ & \bowtie_{rvalue=email\_id}\ employeelist) \\ q_{addressbook} =& \pi_{(rvalue,nickname)}(q_{rec\_eid} \\ & \bowtie_{q_{rec\_eid}.eid=alias.eid}\ alias) \\ vq_{addressbook} =& addressbook\langle q_{addressbook}, \varepsilon\rangle \end{aligned}$$

2. **Intent**: Check if the message X is signed in feature SIGNATURE.
   Query: We use query $q_{signature}$ to projects *is_signed* directly to check if message X is signed or not (*is_signed*).

$$\begin{aligned} q_{signature} =& \pi_{is\_signed}(\sigma_{mid=X}\, messages) \\ vq_{signature} =& signature\langle q_{signature}, \varepsilon\rangle \end{aligned}$$

3. **Intent**: Check if the message X is encrypted in feature ENCRYPTION.
   Query: To express the intent, we project *is_encrypted* for message X from the relation *messages*.

$$\begin{aligned} q_{encryption} =& \pi_{is\_encrypted}(\sigma_{mid=X}\, messages) \\ vq_{encryption} =& encryption\langle q_{encryption}, \varepsilon\rangle \end{aligned}$$

4. **Intent**: Given a message X, return the recipient's autoresponder email in the feature AUTORESPONDER.
   Query: To express the intent, we use query $q_{autoresponder}$ to request the body and subject of auto-reply message for the recipient, where we reuse the sub-query q_{rec_eid} to get the recipient information for message X.

$$\begin{aligned} q_{autoresponder} =& \pi_{(auto\_msg.subject, auto\_msg.body)}(q_{rec\_eid} \\ & \bowtie_{q_{rec\_eid}.eid = auto\_msg.eid} auto\_msg) \\ vq_{autoresponder} =& autoresponder\langle q_{autoresponder}, \varepsilon \rangle \end{aligned}$$

5. **Intent**: Given a message X, return the recipient's forward address in the feature FORWARDMESSAGES.
   Query: To express the intent, we use query $q_{forwardmessages}$ to request the forward address of the recipient. Again, we reuse the sub-query q_{rec_eid} to get the recipient information for message X.

$$\begin{aligned} q_{forwardmessages} =& \pi_{forwardaddr}(q_{rec\_eid} \\ & \bowtie_{q_{rec\_eid}.eid = forward\_msg.eid} forward\_msg) \\ vq_{forwardmessages} =& forwardmessages\langle q_{forwardmessages}, \varepsilon \rangle \end{aligned}$$

6. **Intent**: Given a message X, return the sender's pseudonym in the feature REMAILMESSAGE.
   Query: To express the intent, we use the query $q_{remailmessage}$ to request the sender's pseudonym, where it use sub-query q_{sender_eid} to get the sender's information of message X, and join it with *remail_msg* to return the pseudonym of the sender.

$$\begin{aligned} q_{sender\_eid} =& \pi_{(eid, sender, mid)}((\sigma_{mid=X} messages) \\ & \bowtie_{sender = email\_id} employeelist) \\ q_{remailmessage} =& \pi_{(sender, pseudonym)}(q_{sender\_eid} \\ & \bowtie_{q_{sender\_eid}.eid = remail\_msg.eid} remail\_msg) \\ vq_{remailmessage} =& remailmessage\langle q_{remailmessage}, \varepsilon \rangle \end{aligned}$$

7. **Intent**: Given the email message X, return the recipient's filter suffix in the feature FILTERMESSAGES.
   Query: To express the intent, we use the query $q_{filtermessages}$ to request the provisioned suffix that the recipient wants to filter, where we reuse the sub-query q_{rec_eid} to get the recipient's information of message X and then join it with relation *filtermsg* to get the suffix.

$$\begin{aligned} q_{filtermessages} =& \pi_{sender,suffix}(q_{rec\_eid} \\ & \bowtie_{q_{rec\_eid}.eid=filtermsg.eid} filtermsg) \\ vq_{filtermessages} =& filtermessages\langle q_{filtermessages}, \varepsilon\rangle \end{aligned}$$

8. **Intent**: Given the email message X, return the user-name of the recipient in the feature MAILHOST.
   Query: To express the intent, we use the query $q_{mailhost}$ to request the recipient's user-name in the mailhost. We reuse the sub-query q_{rec_eid} to get the recipient's information of message X and then join it with relation *mailhost* to get the username.

$$\begin{aligned} q_{mailhost} =& \pi_{(rvalue,username,mailhost)}(q_{rec\_eid} \\ & \bowtie_{q_{rec\_eid}.eid=mailhost.eid} mailhost) \\ vq_{mailhost} =& mailhost\langle q_{mailhost}, \varepsilon\rangle \end{aligned}$$

4.4.2 V-Queries for Feature Interactions

In this section, we provide 16 v-queries to inquire the data for producing an email message when we consider fixing the feature interactions in the email SPL. All the feature interactions and their corresponding situations and fixes are taken from [10]. There are a total of 27 interactions in the original work, we only adopt 16 of them because the other 11 interactions either have no fix available or the fixes have no database involved. For each interaction and fix, we use q for a plain query to request data in a single scenario and vq for a v-query which takes variation into account. Note that the $q_{feature}$ and $vq_{feature}$ are referencing the plain queries and v-queries in Section 4.4.1.

1. **Intent**: Fix interaction SIGNATURE vs. FORWARDMESSAGES (1).
   Situation: Suppose Bob sends a signed email to Sam who does not provide a signing key, and then Sam forwards this email to a third party. When the forwarding email arrives at the third party, the SIGNATURE feature cannot verify this signed email, because the verification of signature is determined by the sender of the email. In this case, the sender of this forwarding email is shown as Sam's address in the header, but the email is originally signed by Bob.
   Fix: The fix for this would be altering FORWARDMESSAGES so that it does not change the sender of a signed message in the header. In the example situation, it will not change the sender of that forwarding email to Sam. Instead, it keeps the sender as Bob, so when email arrives of the third party, the SIGNATURE can verify the signature with Bob's signing (private) key.
   Query: As a part of the program developed for SPL products, when generating the forwarding message based on an incoming message X, the query q requests the related data including the sender (captured by the attribute *sender*) and the recipient address (*rvalue*) of message X, the forward address (*forwardaddr*) of the recipient, and the information about whether the message X is signed (*is_signed*). When these two features are enabled, the system checks if the message X is signed by the sender (*is_signed* is `true`) and also the recipient of the message is provisioned with a forward address (*forwardaddr* is not `NULL`). If so, the sender of the forward message should be the original sender (*sender*). Otherwise, it should be the recipient of this message (*rvalue*). Additionally, we also perform the normal query for each feature when 2 features do not interact: we have $q_{signature}$ when SIGNATURE is enabled, and $q_{forwardmessages}$ if SIGNATURE is disabled. Thus, we construct a v-query vq.

$$
\begin{aligned}
q_{join-rec-emp-msg} =& \sigma_{mid=X}((messages \\
& \bowtie_{messages.mid=recipientinfo.mid} recipientinfo) \\
& \bowtie_{rvalue=email\_id} employeelist) \\
q =& \pi_{(sender,rvalue,forwardaddr,is\_signed)}(q_{join-rec-emp-msg} \\
& \bowtie_{q_{join-rec-emp-msg}.eid=forward\_msg.eid} forward\_msg) \\
vq =& signature\langle forwardmessages\langle q, q_{signature}\rangle, q_{forwardmessages}\rangle
\end{aligned}
$$

2. **Intent**: Fix interaction SIGNATURE vs. REMAILMESSAGE.
   Situation: Bob signs an email message and then sends through a remailer to Sam since Bob wants to use a pseudonym in this message. However, when this signed message arrives at Sam's client, he may identify the pseudonym of this message is Bob, since the signature of the message gives a clue Sam.
   Fix: The UI shows a dialog to inform the user that he is sending a signed message to a remailer.
   Query: Given a message X, we have a plain query q to inquire if the message X is signed (*is_signed*) and the recipient of the message (*rvalue*). when these two features are enabled, the system produce the UI to apply the fix when the message X is signed by the user (*is_signed* is `true`) and the recipient of the message X (*rvalue*) is a remailer host. While if one of these two features is disabled, we don't need to query anything. Thus, we construct a v-query vq.

$$q = \pi_{(is\_signed, rvalue)}(\sigma_{mid=X}(messages \bowtie_{messages.mid=recipientinfo.mid} recipientinfo))$$
$$vq = (signature \wedge remailmessage)\langle q, \varepsilon \rangle$$

3. **Intent**: Fix interaction ENCRYPTION vs. AUTORESPONDER.
   Situation: Bob sends an encrypted message to Sam, and Sam has an auto-responder email provisioned. When Sam received the message and decrypts it successfully, his email also auto-responses this message back to Bob, but the header contains the clear information of the original encrypted message. This defeats the privacy purpose of ENCRYPTION, since the information of original encrypted email is exposed in open networks.
   Fix: Make AUTORESPONDER exclude the privacy information in the header when the feature detects that it is auto-replying a decrypted message.
   Query: Given a message X, we use a plain query to inquire related data, it includes the encrypt information (*is_encrypted*) and the recipient (*rvalue*) of message X, and the id, subject, body of autoresponder (*auto_msg.subject* and *auto_msg.body*). When these two features are enabled, the system checks if the incoming message X is encrypted (*is_encrypted* is `true`) and the recipient of message X is provisioned with autoresponder (*auto_msg.eid* is not `NULL`). If so, the system applies the fix and remove the privacy information from the header, that is, the system only need

to generate the subject and body of autoresponder the recipient provisioned.

$$q = \pi_{(is\_encrypted, rvalue, auto\_msg.eid, auto\_msg.subject, auto\_msg.body)}(q_{join-rec-emp-msg} \bowtie_{q_{join-rec-emp-msg}.eid = auto\_msg.eid} auto\_msg)$$
$$vq = encryption\langle autoresponder\langle q, q_{encryption}\rangle, q_{autoresponder}\rangle$$

4. **Intent**: Fix interaction ENCRYPTION vs. FORWARDMESSAGES.
   Situation: Bob sends an encrypted message to Sam, and Sam decrypts it successfully. Since Sam has a FORWARDMESSAGES enabled, the decrypted message is forwarded to a forward-address. This defeats the privacy purpose since the decrypted email travels in open networks during the process of forwarding.
   Fix: Make FORWARDMESSAGES recognize the email that's going to be auto-forwarded is a decrypted one, so instead of forwarding the decrypted message, FORWARDMESSAGES could send a system notification to forward-address, and tell the user to check the email in the old address.
   Query: Given a message X when these two features are enabled, the system checks if the message X is encrypted and the recipient's forward address is also provisioned. If so, the system sends a notification. The information needed in this process includes the encryption status($is\_encrypted$) and recipient($rvalue$) of message X, and the forward address of recipient($forwardaddr$). While when the two features do not interact with each other, the system does the normal query for each enabled feature.

$$q = \pi_{(is\_encrypted, rvalue, forwardaddr)}(q_{join-rec-emp-msg} \bowtie_{q_{join-rec-emp-msg}.eid = forward\_msg.eid} forward\_msg)$$
$$vq = encryption\langle forwardmessages\langle q, q_{encryption}\rangle, q_{forwardmessages}\rangle$$

5. **Intent**: Fix interaction ENCRYPTION vs. REMAILMESSAGE.
   Situation: How REMAILMESSAGE works is that it rewrites the sender with the pseudonym in the header. But if the email is encrypted, the information in the header is no longer visible in the remailer, so the REMAILMESSAGE will not work as expected.

Fix: Make ENCRYPTION detect if the message will be sending to a remailer. If so, remove the sender information from the header.
Query: Given a message X when these two features are enabled, the system checks if message X is encrypted and the recipient of this message is a remailer address. If so, the system removes the sender information of the message X in the header. We use vq to inquire the information needed in this checking process, it includes the encryption information ($is\_encrypted$) about message X, the sender ($sender$) and recipient ($rvalue$) of message X.

$$\begin{aligned} q =& \pi_{(is\_encrypted, sender, rvalue)}(\sigma_{mid=X}(messages \\ & \bowtie_{messages.mid=recipientinfo.mid} recipientinfo)) \\ vq =& encryption\langle remailmessage\langle q, q_{encryption}\rangle, q_{remailmessage}\rangle \end{aligned}$$

6. **Intent**: Fix interaction AUTORESPONDER vs. FORWARDMESSAGES.
Situation: Bob sets auto-forwarding the incoming messages to Sam, but Sam has set an autoresponder email. A third party emails a message to Bob, then it will forward to Sam. At this point, Sam will send back the auto-responder message to Bob, and then Bob forward it to Sam again, leading to a loop between Bob and Sam.
Fix: This could be fixed by manipulating the message header in FORWARDMESSAGES. If these two features interact, every message forwarded from Bob to Sam should have a comprehensive header that contains the original sender. In this case, Sam could auto-reply to the third party, instead of Bob, preventing the loop.
Query: Given a message X when these two features are enabled, the system checks if the message X is an auto-forward message and also the recipient has autoresponder provisioned. If so, the system should modify the message X by adding the original sender in the header. We have a query q requesting the forwarding property of message X, its original sender and recipient, and the autoresponder

provisioned by recipient.

$$q = \pi_{(sender, rvalue, forwardaddr, auto\_msg.eid, auto\_msg.subject, auto\_msg.body)}((q_{join-rec-emp-msg} \bowtie_{q_{join-rec-emp-msg}.eid = auto\_msg.eid} auto\_msg) \bowtie_{q_{join-rec-emp-msg}.eid = forward\_msg.eid} forward\_msg)$$

$$vq = autoresponder\langle forwardmessages\langle q, q_{autoresponder}\rangle, q_{forwardmessages}\rangle$$

7. **Intent**: Fix interaction AUTORESPONDER vs. REMAILMESSAGE (1).
   Situation: Bob sends an anonymous message to Sam through a remailer, and later on, Bob sets AUTORESPONSE when he is out of office. Then Sam replies to Bob's anonymous message which then remailer sends to Bob. There is a problem happens, Bob activates his AUTORESPONDER, so that he auto-replies the message back to Sam, which infers that the previous anonymous message is from Bob.
   Fix: AUTORESPONDER detects if the incoming message is from remailer (it can tell by examining email header), if so, do not auto response to it.
   Query: Given a message X when the two features are enabled, the system checks if the message X is from the remailer, and the recipient of message X has autoresponder provisioned. If so, the system will not auto-reply the message X. We use query q requesting the related data, it includes the sender($sender$) of the message X, and the autoresponder ($auto\_msg.subject$, $auto\_msg.body$) information for the recipient ($rvalue$).

$$q = \pi_{(sender, rvalue, auto\_msg.eid, auto\_msg.subject, auto\_msg.body)}(q_{join-rec-emp-msg} \bowtie_{q_{join-rec-emp-msg}.eid = auto\_msg.eid} auto\_msg)$$

$$vq = autoresponder\langle remailmessage\langle q, q_{autoresponder}\rangle, q_{remailmessage}\rangle$$

8. **Intent**: Fix interaction AUTORESPONDER vs. FILTERMESSAGES.
   Situation: The system provisions FILTERMESSAGES to filter out all the incoming messages from `linkedin.com`. Bob, a user, sends an invitation email to `sam@linkedin.com` to ask him if he can join Sam's network. `Sam@linkedin.com` has activated AUTORESPONDER so he auto-replies the information that he is out of town these days. However, this auto-responder message from Sam has never

arrived in Bob's inbox, because all incoming emails from `linkedin.com` have been filtered out. It defeats the purpose of feature FILTERMESSAGES.
Fix: Make FILTERMESSAGES do not filter the auto-responder email.
Query: Given a message X when the two features are enabled, the system checks if message X is a autoresponder message, if so, the system do not filter the message X. We use query q to request the autoresponder information about message X.

$$q = \pi_{is\_autoresponse}(\sigma_{mid=X}\, messages)$$
$$vq = autoresponder\langle filtermessages\langle q, q_{autoresponder}\rangle, q_{filtermessages}\rangle$$

9. **Intent**: Fix interaction AUTORESPONDER vs. MAILHOST.
Situation: Bob provisions with a auto-responder and then he sends a message to an unknown user. The Mailhost generates a Non-Delivery notification to inform Bob of the non-exists user problem. But, this notification is auto replied by Bob and this auto-responder message is redundant.
Fix: Make AUTORESPONDER identify Non-Delivery notification and do not auto reply.
Query: Given a message X when the two features are enabled, the system checks if the message X is a system notification. If so, the system will not auto reply the message X. We use query q to inquire if the message X is a system notification ($is\_system\_notification$).

$$q = \pi_{is\_system\_notification}(\sigma_{mid=X}\, messages)$$
$$vq = (autoresponder \wedge mailhost)\langle q, \varepsilon\rangle$$

10. **Intent**: Fix interaction FORWARDMESSAGES vs. FORWARDMESSAGES.
Situation: Bob sets forwarding address to Sam, and Sam sets forwarding address to Bob accidentally. This will form a loop between Bob and Sam, once either of them sends an email to each other.
Fix: FORWARDMESSAGES detects the loop and terminate it.
Query: Given a message X when the two features are enabled, the system checks if the sender of the message X is the same with the forward address of the recipient. If so, the system does not forward the message X. We use q to request the data

used in this checking process, it includes the sender (*sender*), the recipient (*rvalue*) and the forward address of the recipient (*forwardaddr*).

$$\begin{aligned} q =& \pi_{(sender,rvalue,forwardaddr)}(q_{join-rec-emp-msg} \\ & \bowtie_{q_{join-rec-emp-msg}.eid=forward\_msg.eid} forward\_msg) \\ vq =& forwardmessages\langle q, \varepsilon\rangle \end{aligned}$$

11. **Intent**: Fix interaction FORWARDMESSAGES vs. REMAILMESSAGE (1).
Situation: Bob sets forwarding address to his pseudonym in the remailer. This will form a loop between Bob and the remailer if there is a message sending to Bob, because Bob's email will forward to the pseudonym in the remailer and then the message will loop back from remailer and forth infinitely.
Fix: Make feature REMAILMESSAGE detect the loop and terminate it.
Query: We use q to request the data used to check the forward Address and pseudonym for each user (remailer detect the loop). If the forward address is the same with the pseudonym, the system should terminate the loop.

$$\begin{aligned} temp =& employeelist \bowtie_{employeelist.eid=forward\_msg.eid} forward\_msg \\ q =& \pi_{(email\_id,forwardaddr,pseudonym)}(temp \bowtie_{temp.eid=remail\_msg.eid} remail\_msg) \\ vq =& (remailmessage \wedge forwardmessages)\langle q, \varepsilon\rangle \end{aligned}$$

12. **Intent**: Fix interaction FORWARDMESSAGES vs. FILTERMESSAGES.
Situation: Bob sets forwarding message to `sam@pgn.com`, but he doesn't know pgn's admins have already set filter suffixes that discard all emails from Bob's domain. This lead to that Bob's all incoming emails disappear after that.
Fix: This could be fixed by sending a verification email from Bob to forward address when Bob set his forward email at the first time.
Query: When the user X sets the forward address in a system with the two features are enabled, the system sends a same test message to the user's forward address.

We use query q to return the user X's forward address ($forwardaddr$).

$$q = \pi_{forwardaddr}(\sigma_{eid=X}(employeelist \bowtie_{employeelist.eid=forward\_msg.eid} forward\_msg))$$
$$vq = (forwardmessages \wedge filtermessages)\langle q, \varepsilon \rangle$$

13. **Intent**:Fix interaction FORWARDMESSAGES vs. MAILHOST.
Situation: Bob sets forward address to a non-existent user in his mailhost. Any email sent to Bob will forward to the mailhost and the mailhost will send back an error message tell Bob that the address does not existed. The error message sending to Bob will result in an infinite loop between Bob and the mailhost.
Fix: MAILHOST detects the loop and terminates it.
Query: Given a message X sending to a mailhost when these two features are enabled, the system check if the recipient of message X is a user in the mailhost, if its not and the message x is a auto forward one, the mailhost does not generate error message back. We use a plain query q to request the related data, it includes the recipient ($rvalue$) of message X, the user name in the mailhost, and if the message X is an auto forward message ($is\_forward\_msg$).

$$\begin{aligned} q = &\pi_{(rvalue, username, is\_forward\_msg)}(q_{join-rec-emp-msg} \\ &\bowtie_{q_{join-rec-emp-msg}.eid=mailhost.eid} mailhost) \\ vq = &(forwardmessages \wedge mailhost)\langle q, \varepsilon \rangle \end{aligned}$$

14. **Intent**:Fix interaction REMAILMESSAGE vs. MAILHOST
Situation: Bob activates mailhost and his remailer service, then he use email at mailhost to send an anonymous email to Sam. After a while, Bob cancel his mailhost service but forget to deactivate the remailer pseudonym. Then, Sam replies to the pseudonym address, and the remailer deliver the message to the Bob's address at mailhost. Since Bob's address is not a user existing in the mailhost, the mailhost generates an automated message back, leaking the identity of Bob.
Fix: Make MAILHOST detects email from remailer and send back a Non-delivery Notification without containing user's information if applicable.
Query: Given a message X sending to a mailhost when the two features are enabled,

the system checks if the sender of the message is from a remailer (The mailhost can tell if it's from remailer by checking the sender's email address), if so, the mailhost generates reply message without containing user's information. We use q to inquire the sender's email address.

$$q = \pi_{sender}(\sigma_{mid=X}\, messages)$$
$$vq = (remailmessage \wedge mailhost)\langle q, \varepsilon\rangle$$

15. **Intent**: Fix interaction FILTERMESSAGES vs. MAILHOST.
Situation: Bob activates FILTERMESSAGES to filter out all the messages from domain `outlet`. Bob sends an email to `non-existed-user@outlet.com` which has no matched user-name found in that domain. Then the mailhost service in the `outlet` domain generates an automated system notification back to Bob which informs him that no user is found. However, FILTERMESSAGES in Bob side discards such notification which defeats the purpose of MAILHOST.
Fix: Alter FILTERMESSAGES check if the incoming system notification is from a mailhost and also this message corresponds to an appropriate outbound message.
Query: Given a message X sending from a mailhost, when these two features are enabled, the system check if the message X is a system notification, if so, the system doesn't need to check the filter suffix and let the message X delivered. We use q to request the data including the sender of the message ($sender$) and if the message X is a system notification ($is\_system\_notification$).

$$q = \pi_{(is\_system\_notification, sender)}(\sigma_{mid=X}\, messages)$$
$$vq = (filtermessages \wedge mailhost)\langle q, \varepsilon\rangle$$

16. **Intent**: Fix interaction SIGNATURE vs. FORWARDMESSAGES (2).
Situation: Suppose Bob sends a signed email to Sam, and Sam verifies the signature then auto-forward to a third party with a "success-verified" header. In the process of forward after verification, the message may be altered by hacker during the transit, because the verified email is in a clear form. However, the recipient of this email thinks this verified email is trustful because of the "success-verified" header. This defeats the purpose of privacy regulation.

Fix: Alter the feature SIGNATURE to add a header `Verified-at:` which informs the recipient where the verification was done. The recipient should only fully trust the email verified at their own end.
Query: Given a message X when these two features are enabled, the system check if the message X is signed and its recipient has an auto-forward address provisioned. If so, when forwarding the message X, the system will add the information about where the signature is verified and forward it to forward address. During this process, we use query q to inquire if the message X is signed (is_signed), the recipient ($rvalue$) and its forward address ($forwardaddr$).

$$\begin{aligned} q =& \pi_{(is\_signed, rvalue, forwardaddr)}(q_{join-rec-emp-msg} \\ & \bowtie_{q_{join-rec-emp-msg}.eid = forward\_msg.eid} forward\_msg) \\ vq =& signature\langle forwardmessages\langle q, q_{signature}\rangle, q_{forwardmessages}\rangle \end{aligned}$$

Chapter 5 Conclusion and Future Work

This paper has presented two case studies that investigate the application of VDBMS to schema evolution and SPLs. The first case study demonstrated the ability of VDBMS to archive a historical database and query multiple snapshots of an evolving schema in a single VDB, where we systematically adopted a schema evolution scenario in an existing employee database and generated a VDB from a widely used employee data set. The second case study represented the integration of the VDBMS to an Email SPL, where we systematically adopted the prior work of a feature list and feature interactions in the email system and combined it with a real-world data set from the Enron Corp to generate a VDB for the email SPL.

Each case study has provided a VDB and a set of v-queries. The process of generating the VDB and v-queries gives the VDBMS user a practical tutorial to follow, and the resulting data sets and the example v-queries will be used to evaluate the VDBMS.

In the current iteration of case studies, we wrote the most intuitive v-queries to express the intents, but in the future we would like to apply some optimizing rules manually to v-queries, or implement a component in the VDBMS architecture to achieve the goal automatically.

Bibliography

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

[2] Parisa Ataei, Qiaoran Li, Eric Walkingshaw, and Arash Termehchy. Managing variability in relational databases by vdbms. unpublished draft.

[3] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. Variational databases. In *Proceedings of The 16th International Symposium on Database Programming Languages*, DBPL '17, pages 11:1–11:4, New York, NY, USA, 2017. ACM.

[4] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. Managing structurally heterogeneous databases in software product lines. In Vijay Gadepally, Timothy Mattson, Michael Stonebraker, Fusheng Wang, Gang Luo, and George Teodoro, editors, *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, pages 68–77, Cham, 2019. Springer International Publishing.

[5] M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 1st edition, 2000.

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

[7] Ahmed Elmagarmid, Marek Rusinkiewicz, and Amit Sheth, editors. *Management of Heterogeneous and Autonomous Database Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[8] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, December 2011.

[9] Martin Erwig, Eric Walkingshaw, and Sheng Chen. An abstract representation of variational graphs. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, FOSD '13, pages 25–32, New York, NY, USA, 2013. ACM.

[10] Robert J. Hall. Fundamental nonmodularity in electronic mail. *Automated Software Engineering*, 12(1):41–79, Jan 2005.

[11] 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.

[12] J. Postel. Simple mail transfer protocol, 1982.

[13] Sudha Ram and Ganesan Shankaranarayanan. Research issues in database schema evolution: the road not taken. In *Working Paper 2003-15*. Information Systems Department, Boston University School of Management, 2003.

[14] John F. Roddick. Schema evolution in database systems: An annotated bibliography. *SIGMOD Rec.*, 21(4):35–40, December 1992.

[15] Jitesh Shetty and Jafar Adibi. The enron email dataset database schema and brief statistical report. *Information sciences institute technical report, University of Southern California*, 4(1):120–128, 2004.

[16] Ben Shneiderman and Glenn Thomas. An architecture for automatic relational database sytem conversion. *ACM Trans. Database Syst.*, 7(2):235–257, June 1982.

[17] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational data structures: Exploring tradeoffs in computing with variability. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 213–226, New York, NY, USA, 2014. ACM.