AN ABSTRACT OF THE THESIS OF

Parisa S. Ataei for the degree of Master of Science in Computer Science presented on December 6, 2017.

Title: Principles of Variational Databases

Abstract approved: ______________________________

Arash Termehchy

Data variations are prevalent in real-world applications. For example, software vendors have to handle numerous variations in the business requirements, conventions, and environmental settings of a software product. In database-backed software, the database of each version may have a different schema and content. As another example, data scientists often need to use a subset of the available databases because using non-relevant information may reduce the effectiveness of the results. Such variations give rise to numerous data variants in these applications. Users often would like to query and/or analyze all such variants simultaneously. For example, a software vendor would like to perform common tests over all versions of its product and a data scientist would like to find the subset of information over which the analytics algorithm delivers the most accurate results. Currently, there is not any systematic and principled approach to managing and querying data variations and users have to use their intuition to perform such analyses. We propose a novel abstraction called a *variational database* that provides a compact and structured representation of general forms of data variations for relational databases. As opposed to data integration approaches that provide a unified representation of all data sources, variational databases make variations explicit in both the schema definition and the query language without introducing too much complexity.

Principles of Variational Databases

by

Parisa S. Ataei

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 6, 2017
Commencement June 2018

Master of Science thesis of Parisa S. Ataei presented on December 6, 2017.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

Parisa S. Ataei, Author

ACKNOWLEDGEMENTS

I would first like to thank my adviser, Dr. Arash Termehchy, for providing me with a chance to work on a problem that I am passionate about. Thank you for steering me into the right direction without depriving me of the sense of ownership. Thank you for being thorough and strict when I hoped you would not! And thank you for providing me with endless professional opportunities and guidance.

I would also like to thank my current committee member and co-adviser to be, Dr. Eric Walkingshaw, for all his constructive feedback and encouragement. Thank you for awakening my instinctive interest in programming languages, for answering all my simple stupid questions patiently, for understanding my half-mentioned explanations and paraphrasing them, and for reviewing this thesis with me.

I am also grateful to my committee members and GCR, Prof. Martin Erwig, Prof. Alan Fern, and Prof. Abi Farsoni, for their insightful comments and feedback.

With a special mention to Ben McCamish, thank you for the positive words and encouragement during my most stressful self; Mahtab Aboufazeli, thank you for putting up with my most annoying pet peeves; Arezoo Rajabi, thank you for the valuable chats, studies, and rides; and the CrossFit Train 97333 and its amazing coaches, thank you for teaching me that I can always push past the boundaries that I have set for myself and proving that I am mentally much stronger than what I can even imagine.

And finally, at last but by no means least, my very special gratitude goes out to my parents for providing me with continuous support and encouragement throughout my years of study. Thank you for standing by my side throughout all my decisions that have got me here. This accomplishment would not have been possible without you. Thank you!

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

Variations in the content and representation of databases are common in real-world applications. Users often would like to express the same information need over all database variants in an application. For example, to cope with variations in business requirements, regional conventions, and/or environmental settings for different groups of users, a software company creates different variants of its software products [3]. These variants generally share a common codebase and differ based on the selection of *features* that extend or modify the core functionality. For example, one feature may determine the unit of currency and another one may indicate whether a person must have a unique social security number. A software product may have hundreds of features whose combinations create a large number of software variants. In database-backed software products, each variant may have a different database with distinct schema and content. As each data element may vary in terms of content and representation, there are usually numerous possible database variants in a software product. Software companies often would like to perform some common tests over all these variants, for example, to check that each relation in a relational database has a primary key.

As another example of computing over variations of data, during the process of feature extraction and selection, a data scientist might train a model over many variants of a dataset to find the variant over which the model has the highest accuracy. Since each element may have a distinct representation in or be absent from a variant, the training is done over numerous variations of the underlying dataset. In these applications, users need to query or operate on numerous variants of a dataset conceptually simultaneously.

As a result, users would like to work with a database system that hides and/or simplifies the variational nature of the data. Instead of dealing with myriads of databases with different contents and/or structure, users prefer to work with a unified, compact, and simple representation of these variants. Otherwise, users have to rewrite and reconfigure their queries and algorithms over each variant, which takes a great deal of time and effort. Furthermore, each variant may produce a different answer to the submitted query. Hence, queries written in the query language L over such abstraction must be

country	*member* table
US	$member(ID, firstName, middleName, lastName)$
Iran	$member(ID, firstName, lastName)$
Iceland	$member(ID, firstName, middleName, fatherName, gender)$

Table 1.1: Three different variations of a table in a database-backed software company, where the feature "country" determines the structure of the table.

closed under L: their inputs and outputs are sets of database variants. Example 1.0.1 demonstrates the need for a system that can query multiple databases given only one query.

Example 1.0.1 *Assume a software company that develops variants of software for financial institutions all around the globe. The feature "country" determines the country that software has been requested for. Different countries practice different naming conventions for their customers. The software company has to apply this in their database design. For example, while it is usual to have a middle name in the United States, it is not in Iran. Also, the naming convention in Iceland differs from most Western family name systems by being patronymic, i.e., they indicate the father of the child and not the historic family lineage. For example, if a father's name is "Jón Einarsson", his daughter and son would take the last name "Jónsdóttir" and "Jónsson", respectively. Table 1.1 shows the different designs of one table called "member" for different settings of the feature "country". Assume a developer wants to get the full name of members across all variations for a test. They have to write three different queries: First, for the US's variant, the query should get the first name, middle name, and last name attributes. Second, for Iran's case, it should return the first name and last name. Finally, for Iceland, it should return the first name and father's name attached to "dóttir' or "sson" if the person is female or male, respectively. Rewriting a query as such is a burdensome task for the user as the number of variants can grow exponentially in terms of the number of features. The question becomes: is there a way that the user can just write one query for querying all variants of a database?*

Views have traditionally been used to define unified abstractions over multiple databases. Nevertheless, the result of a query over views is a single table. As shown in Example 1.0.1 in such applications we need to be able to output a set of tables depending on the underlying schema of the result. Moreover, due to the possible variability of each element in the underlying dataset (e.g., in a database-backed software product line), one may face exponentially many variants in an application. It is not clear how to scale a view definition to this many databases.

In this thesis, we outline our proposal for a novel abstraction called *variational databases*, v-DB for short, that provides a unified, compact, and structured representation of data variants in an application domain. We focus on the application of software product lines. We also define the concept of *variational queries* whose inputs and outputs are variational as well. One may use variational queries to explore database variants simultaneously while preserving differences between them. Here, we mainly focus on schema variations in data. However, we believe that variational databases have the potential to handle other types of data variations.

In Chapter 2 we give a comprehensive background of the situations where more than two databases are used at a time and how people handle them depending on their application. Then in Chapter 3 we provide some preliminary definitions along with the used feature model and its properties (Chapter 4), we formalize *variational databases* in Chapter 5, and introduce the representation system, called *v-table*, for it in Section 5.3. After that, we introduce our query language, called *variational queries*, and define its syntax and semantics in Chapter 6. Finally, we conclude in Chapter 7 and outline future work.

Chapter 2: Literature Review

To the best of our knowledge, there exists no database system and query language that supports accessing and manipulating multiple databases with different schemas simultaneously. This is a crucial issue in the database-backed software generated by software product lines (SPL). Although there has been little attention to the database variability related to these software variations. Mathieu et al. present methods to generate the database of different variations [13, 12], as opposed to a variational database that assumes the database of variations has been provided and tackles the consequences of working with various schemas simultaneously. Khedri and Khosravi propose a delta-oriented programming approach that uses a feature dependency graph of an SPL to generate the data definition language (DDL) scripts of database-backed software automatically [13]. In contrast, Mathieu et al. present a plug-in tool for the DB-Main CASE tool, a tool that supports management of database-related variability. They base their work on feature-oriented modeling analysis and the Simple Variability Language [12].

However, the problem with these techniques is twofold: First, they focus on a database DDL and do not consider the effects of variability on other aspects such as queries used in the implementation of SPL software. Second, they are limited to the context of database variations in SPLs and cannot directly be extended to other applications. In contrast, variational databases seek to model general forms of data variations. For example, experts of a data analytics system generate variations of a database to evaluate their system on different schemas [16]. Moreover, while some approaches offer a representation-independent relational learning algorithm such as Castor, others generate different results with regards to the underlying schema of the data such as FOIL and ProGolem [16]. Variational databases and variational queries provide a language that can first simplify the expert's job of creating and querying variations of a database and second be used to modify such algorithms slightly to be executable over multiple variations of the data and pick the best result possible.

The dependency of queries on the layout of data has been studied extensively for various data formats. While this dependency is apparent for some structured data such as relational, it is not very obvious for others like HTML pages. Dalvi et al. use training examples to learn a tree-based model and intuitive heuristics to achieve a more robust XPath query over similar HTML pages [6]. They also introduce two types of robustness, adversarial and probabilistic, and use learning algorithms to learn two different models to achieve robust XPath queries over a series of HTML pages that have their layouts changed over time [15]. Omari et al. use annotated pages and modify the XPath query language [14]. They then generate a decision tree to obtain a robust query with a precision that adjusts dynamically. These approaches use training examples and learn some model to achieve a robust query against schema modifications, as opposed to a variational database that does not use any annotated data to achieve executable queries against various representation of data. Instead, we provide an expressive language that encodes a query that can be applied to multiple variations of a database. Note that we do not introduce a robust query against the representation of data. We rather introduce a query language that is capable of carrying the representation of data inside it.

The query dependency on the representation of the structured data becomes more problematic when the database evolves throughout time, known as **database evolution**. Schema evolution tools try to facilitate the database administrator (DBA) task of generating a new database, assuming that the DBA knows the target schema exactly [5]. More specifically, they assist the DBA with the process of generating the new database, determining if it is information preserving, transforming queries written over the previous schema to the new one, and keeping a history of steps taken to generate the new database in order to be able to reverse it, if chosen by the user. In short, they provide the users and DBAs with a language to express the mapping between a schema and its evolved version and translating queries given a specific mapping [10, 5]. Furthermore, they can neither execute a query over multiple versions of a database nor are they able to return a set of results to answer a query evaluated on numerous versions. Nevertheless, variational databases aim at a more convenient and semi-automatic query maintenance in the face of various categories of schema evolution and mapping.

In addition to schema evolution, the need to query different data sources concerns **data integration**. Data integration is the gathering of data of different formats from various sources due to a new information need of a company, research group, etc [7]. Due to the vast amount of variety in data, a mediated schema or warehouse is introduced to the architecture of a data integration system such that it can be mapped accordingly to each of the data sources. As a result, queries are written in terms of the mediated schema and will be translated to appropriate queries over relevant data sources using the mappings that have been defined over data sources and the mediated schema, as opposed to data integration, variational databases do not generate a mediated schema with mappings to variations accordingly. Instead, they provide a compact representation of all schemas in an efficient manner such that the user can query each of the desired variations or a number of them. Furthermore, while data integration systems deal with managing heterogeneity among existing data sources, variational databases consider heterogeneous data in a domain of interest, e.g., SPL. As a result, variational databases can encode differences among various schemas easily without explicitly mapping an intermediate schema to the source schemas. Instead, the feature model presented in a variational database links a specific schema design to a variation. Hence, the existence of feature model relieves a variational database from having to define explicit schema mappings.

Similar to the need of defining numerous valid variations of a database, is the need to represent uncertainty in the data, which is the purpose of **probabilistic and incomplete databases**. Probabilistic databases represent uncertainty about the content of a database by maintaining a set of possible states where each state has a probability of being true [17]. These states all share the same schema, and it is assumed that one is the actual state of the database. Variational databases, however, aim at providing a principled approach to modeling general types of variations where there may be numerous valid variations with different schemas. Note that the variability that we consider is different than current uncertain databases such as incomplete (probabilistic) databases. The latter arises when we have incomplete (uncertain) knowledge [2, 17], while the former is the result of information and system need of users to work on different variations of a database at a time. While incomplete (probabilistic) databases engender multiple *possible worlds*, the variational databases introduce just one possible world with variations in data that the user needs to keep track.

In this thesis, we expand the idea of variational databases we introduced in [4]. However, our encoding differs significantly from the previous work on variational schemas. In [4] we present an encoding based on presence conditions and variational sets and maps, similar to previous research on variational lists, sets, and graphs [19, 9]. This approach differs from previous methods of encoding variation in abstract syntax trees [8, 18] with *choices*. A choice $f\langle e_1, e_2\rangle$ is labeled by a condition f and resolves to either the alternative e_1 if f is `true` or e_2 if f is `false`. Nonetheless, we based our representation in this work on formula choice calculus (FCC) [11], which is established over choices. This representation allows for a closer interaction with relational algebra. Furthermore, employing FCC permits us to handle the feature sets introduced by the application of interest, SPL. It enables the variational database to encode various elements of a kind in one including schemas and queries. Nevertheless, the representation of variational queries and schemas in both works differs.

Chapter 3: Preliminaries

In this section, we first review some basic database concepts required. Then we introduce the concept of Software Product Line (SPL) and how it relates to variational databases. Schema S is a finite set $\{R_1, \dots, R_n\}$ of relation symbols where each R_i has a fixed arity $n_i \geq 0$. A relation R_i consists of an attribute set, i.e., $attr(R_i) = \{A_1, \dots, A_k\}$. The *cardinality* of a relation, denoted by $arity(R_i)$, is the number of attributes it contains. Let D be a countably infinite set of constants. An instance I_S of S assigns to each relation symbol $R_i \in S$ a finite n_i-ary relation $R_i^I \subseteq D^{n_i}$. For brevity, we use I to denote an instance of a database. The domain $dom(I)$ of instance I is the set of all constants that occur in at least one relation in I. In a defined instance I, each attribute also has a domain, denoted by $dom(A_i)$, defined as the set of all constants assigned to A_i in instance I.

Software saturates every sector. Organizations rely on its effectiveness and efficiency every day. Hence, there is a considerable demand for similar software all around the globe. As a result, software vendors produce families of similar systems that differentiate by features [1]. Such software primarily performs the same or similar tasks. Developing a software requires a certain amount of time and effort, not to mention the resources it occupies for the maintenance. Hence, if there was a way to generate families of software that conduct similar tasks without the need to develop each one from scratch, we could have saved a lot of resources: time, effort, and money! Developing a core asset combined with a product development and management is the primary idea of *Software Product Lines*, SPL for short. SPL is an approach for producing software-intensive products [1]. A software product practice takes advantage of a core asset and applies numerous features in the product development to generate various software that conducts tasks in the same category.

Example 3.0.1 *A software product practice uses a pipeline to generate software products of banking, stock marketing, and investment, since they all satisfy the need of financial market segment and can be developed from a common set of core assets in a prescribed way [1].*

An SPL practice can generate different software without developing every single one of them by introducing a set of features, called *feature model.* The core asset of an SPL approach is written in such a way that it can decide how to generate a product given a set of configured features. We call each of the products created this way a *variation.* In Chapter 4 we elaborate more on how features are configured and how we specify them for our purposes.

Often, variations generated by an SPL practice need to work with a database system to be able to store and manipulate their data in a structured way, especially since they need to handle a vast amount of data most of the times. As a result, a schema must be defined for each variation. For our purposes, we introduce some new concepts to be able to refactor most of the database-related work of an SPL.

Definition 3.0.2 (Complete set of domain relations) *Consider a set of countable and finite relation symbols denoted by $\mathcal{R} = R_1, \ldots, R_n$. Let $\mathcal{A}_i = \{A_1, \ldots, A_k\}, 1 \leq i \leq n, k = arity(R_i)$ be a disjoint countable and finite set of attributes. We call each of $R_i(A_1, \cdots, A_k)$ for $1 \leq i \leq n$ a* **domain relation** *and the whole set of them a* **complete set of domain relations***.*

We use the term *SPL relation* interchangeably with *domain relation* here since we focus on the application of SPLs. A schema design may use a subset of the SPL relations and subsets of their attributes, resulting in a *schema variation.* We limit the transformation that schema variations can take into two types: 1) they can contain a relation defined in the complete set of domain relations or not and 2) a relation of a variation can include an attribute of that domain relation or not. This assumption requires attribute and relation names to be distinct and unique, i.e., we cannot have two attributes (relation) representing semantically same concepts.

Definition 3.0.3 (Possible relations) *A set of relations in different variations that are instances of the relation R, defined as an SPL relation, is called a set of* **possible relations** *of R.*

Here we define our problems in details: *Given a software product practice (including its variations) along with a set of schema variations and their associated feature configurations, how can a user query all the variations at the same time and get the results over all possible variations?*

Chapter 4: Feature Expression

As mentioned earlier we allow two types of transformations among variations of a software: 1) including a table or not and 2) including an attribute or not. The inclusion of a relation or an attribute is determined by the configured features for every variation. In this chapter, we first explain the feature model used in an SPL and how we use feature expressions to specify the SPL feature model. We then explain the equivalence of two feature formulas, which will be used for evaluating queries.

4.1 Feature Expressions Specifying the SPL Feature Model

As mentioned earlier, an SPL uses a set of features, denoted by **F**, to manage different variations of a software. We assume that features are boolean variables. We show in Example 4.1.1 that we can easily convert multi-valued features to boolean ones. Hence, without loss of generality, we assume that features are booleans for now. We consider two types of feature formulas: 1) clauses that only conjunct features (literals) together and 2) disjunctions of the conjunctive clauses. Each of these cases has its own applications, which we will explain when we use it. We limit our abstract syntax to *DNF* feature formulas. However, we allow unnormalized feature formulas in the concrete syntax. Note that each set of features are associated with a category of features. For example, if we have features *US* and *Iceland* we assign both of them to the category of *country*. We assume this task is done by an SPL expert who is familiar with the feature model of the SPL core. In Example 4.1.1 we show that this is a simple task that in most of the SPLs is not needed due to having multi-valued features and the process of converting them to boolean variables.

Example 4.1.1 *Assume we have the SPL introduced in Chapter 1. We mentioned that it has a feature country that takes three values: United State, Iran, and Iceland. We convert this feature into boolean variables: US, Ir, and Ic; denoting countries United State, Iran, and Iceland, respectively. Note that these three boolean features belong to the feature category "country".*

Feature expression generic object:

$$f \in \mathbf{F} \quad \textit{Feature Name}$$

Feature expression syntax:

$$\begin{array}{rcll}
b \in \mathbb{B} & ::= & \texttt{true} \mid \texttt{false} & \textit{Boolean Tag} \\
cl \in \mathbf{Conj} & ::= & b \mid f \mid \neg f \mid cl \wedge cl & \textit{Conjunction of Features} \\
\widehat{f} \in \mathcal{F} & ::= & cl \mid cl \vee cl & \textit{Feature Formula} \\
c \in \mathbf{C} & = & \mathbf{F} \rightarrow \mathbb{B} & \textit{Configuration}
\end{array}$$

Semantics of feature formulas:

$$\begin{aligned}
F[\![.]\!] &: \mathcal{F} \rightarrow \mathbf{C} \rightarrow \mathbb{B} \\
F[\![b]\!]_c &= b \\
F[\![f]\!]_c &= c\ f \\
F[\![\neg f]\!]_c &= \neg F[\![f]\!]_c \\
F[\![cl_1 \wedge cl_2]\!]_c &= F[\![cl_1]\!]_c \wedge F[\![cl_2]\!]_c \\
F[\![cl_1 \vee cl_2]\!]_c &= F[\![cl_1]\!]_c \vee F[\![cl_2]\!]_c
\end{aligned}$$

Figure 4.1: Feature expression definition.

As we mentioned earlier, the configured features determine the specific variation. Hence, we define a *configuration* as a function that maps every feature of the SPL to a boolean tag. By definition, a configuration is a total function from the feature set to boolean tags, i.e., a configuration includes all features defined in the SPL feature model. For brevity, we show the configuration c with a set of features, denoted by F_c^t, where c maps all features in F_c^t to `true`, i.e., $\forall f \in F_c^t : F[\![f]\!]_c = \texttt{true}$. Every other feature in the SPL feature model that is not included in F_c^t is set to `false` under configuration c, i.e., $\forall f \notin F_c^t : F[\![f]\!]_c = \texttt{false}$. We call F_c^t the *true feature set* of configuration c.

Example 4.1.2 *Assume the SPL used in Example 4.1.1 also has the feature category type with boolean variables b, s, and i which represent banking, stock marketing, and investment, respectively. These features determine the type of the software. Now the software requested for banking in the United State has the configuration c s.t.:* $F_c^t = \{US, b\}$*. Every other feature is set to* `false`*, i.e., Ir, Ic, i, and s are set to* `false` *under configuration c.*

4.2 Feature Equivalence

In this section, we establish syntactic rules for deriving the semantic equivalence of feature formulas. We will see in Section 6.4 that this relation is required for the term equivalence of our query language. In practice, formula equivalence can be checked using a SAT solver. Recall that

1. Two functions are equal, by definition, if they have the same domain and codomain and their images agree for all elements in the domain.
2. For any function, the inverse images of elements in the codomain partition the domain.
3. Any partition defines a canonical equivalence relation where the parts of the partition correspond to the equivalence classes of the relation.

Definition 4.2.1 (Feature formula equivalence) *Let* $(\equiv)$ *be a binary relation on feature formulas defined by* $\widehat{f} \equiv \widehat{f'}$ *iff* $\forall c \in \mathbf{C} : F[\![\widehat{f}]\!]_c = F[\![\widehat{f'}]\!]_c$.

Note that the feature formula equivalence is defined in terms of function equality. Since $F[\![.]\!]$ is a well-defined function from feature formulas to elements of the semantic domain, it follows from earlier remarks that the relation $(\equiv)$ is an equivalence relation. We refer to the relation $(\equiv)$ as a *semantic equivalence* and we call feature formulas $\widehat{f}$ and $\widehat{f}'$ *(semantically) equivalent* if $\widehat{f} \equiv \widehat{f}'$. For a set of feature formula equivalence rules, the reader can consult [11].

Chapter 5: Variational Databases Definition and Formalization

A v-DB, which stands for *variational database*, is intuitively meaningful when we have a set of *schema variations* at the same time and we want to work on some of them simultaneously. We consider a schema variation, S, *valid* if there exists a software product with a database instance employing schema S in the underlying SPL practice. In this chapter, we define a compact representation of a set of valid schemas, called a *variational schema*. We also elaborate on the equivalence of the elements of such a schema in Section 5.2. We then establish a representation system for variational databases in Section 5.3.

5.1 Variational Schema (V-Schema)

Intuitively a *v-schema*, short for *variational schema*, is a systematic and compact representation of all valid schemas of the underlying application of interest that encodes feature-related information effectively. A v-schema relieves the need to define an intermediate schema and state mappings between it and source schemas. A naive representation of variational schemas would be to make the set of relation specifications variational. While this allows arbitrary variation among schemas, it does not reflect the fact that schemas can vary in systematic ways. For example, generally, we want to support any variant that includes or excludes a relation or an attribute. Expressing all such inclusions/exclusions explicitly and one-by-one is tedious and obfuscates more interesting variations. Instead, we encode relations and schemas by using feature expressions defined in Chapter 4 and adopting the concept of formula choice calculus introduced in [11].

Definition 5.1.1 (Variational set) *As shown in Figure 5.1, a* **variational set** *can be a single attribute, a set of attributes, a choice of two variational attribute sets where a feature formula determines which set to pick, or an empty set denoted by* $\varnothing$*.*

Example 5.1.2 *Assume we are given a domain relation deposit($account$, $amount$, $depositor$), and the feature category "type" that contains three boolean variables b, s, and i, representing types banking, stock marketing, and investment, respectively. If the software is designed for a stock market company, then it does not need to have any of the attributes in the deposit relation. However, if it is of the other two types, it will have attributes account and amount. Although it will only contain the "depositor" attribute if it is banking software. Since a person other than the owner of an account can deposit money to that account, which is not the case for investment software. We can show all these variations in the following variational set.*

$$vl_{deposit} = (b \wedge \neg s \wedge \neg i)\langle\{account, amount, depositor\}, (\neg b \wedge \neg s \wedge i)\langle\{account, amount\}, \varnothing\rangle\rangle$$

Definition 5.1.3 (Variational relation) *As shown in Figure 5.1, a* **variational relation** *is a relation with a variational set assigned to it. Note that the feature formulas used in the variational set assigned to the variational relation determine the set of possible relations of a domain relation.*

Example 5.1.4 *We can show the deposit relation with conditions mentioned in Example 5.1.2 by:* $[deposit : vl_{deposit}]$

Definition 5.1.5 (V-schema) *As shown in Figure 5.1 a* **v-schema** *is a choice of "a set of variational relations" with a feature formula attached to it that determines the schema of the database instance for a software product with a configuration in the underlying SPL practice, and an empty set, denoted by $\varnothing$, specifying the absence of a schema for configurations that evaluate the feature formula attached to the choice to* `false`.

Remark 5.1.6 *The attached feature formula, $\widehat{f_s}$, to the choice of a v-schema definition must have the DNF form. As opposed to the feature formula, $\widehat{f_l}$, attached to a choice of variational set that can take any of the forms allowed by feature expression syntax shown in Figure 4.1. This difference is rooted at the purposes that these feature formulas carry. The feature formula $\widehat{f_s}$ indicates all valid schemas. However, the feature formula $\widehat{f_l}$ indicates a subset of the possible relations associated with each set alternative. Therefore, each conjunctive clause of v-schema's feature formula must include all features of the SPL, where each conjunctive clause demonstrates a variation. In contrary, the feature*

Relational model generic objects:

$$\begin{array}{rl} D \in \mathbf{Dom} & \textit{Domain} \\ A \in \mathbf{Att} & \textit{Attribute Name} \\ R \in \mathbf{R} & \textit{Relation Name} \\ t \in \mathbf{T} & \textit{Tuple} \end{array}$$

Relational model definition:

$$\begin{array}{rcll} l \in \mathbf{L} & = & A_1, \ldots, A_n & \textit{Attribute set} \\ s \in \mathbb{S} & = & R(A_1, \ldots, A_n) & \textit{Relation specification} \\ S \in \mathbf{S} & ::= & s_1, \ldots, s_n & \textit{Schema} \\ T \in \mathbf{T} & ::= & \{\langle\!\langle t(1), \ldots, t(k) \rangle\!\rangle | t(i) \in D_i, 1 \leq i \leq k, k = arity(R)\} & \textit{Relation Instance (Table)} \end{array}$$

Variational schema objects:

$$\begin{array}{rcl} vl \in \mathbf{Vset} & ::= & A \ \mid \ vl \cup vl \ \mid \ \widehat{f}\langle vl, vl \rangle \ \mid \ \varnothing \\ VR \in \mathbf{Vrel} & ::= & [R : vl] \end{array}$$

Variational schema syntax:

$$\mathcal{S} \in \mathbf{Vsch} \quad ::= \quad \widehat{f}\langle \{VR_1; \ldots; VR_n\}, \varnothing \rangle$$

Semantics of variational objects:

$$\begin{array}{ll} O_L[\![.]\!] : \mathbf{Vset} \rightarrow \mathbf{C} \rightarrow \mathbf{L} & \\ O_L[\![A]\!]_c & = \{A\} \\ O_L[\![vl_1 \cup vl_2]\!]_c & = O_L[\![vl_1]\!]_c \cup O_L[\![vl_2]\!]_c \\ O_L[\![\widehat{f}\langle vl_1, vl_2 \rangle]\!]_c & = \begin{cases} O_L[\![vl_1]\!]_c, & \text{if } F[\![\widehat{f}]\!]_c = \texttt{true} \\ O_L[\![vl_2]\!]_c, & \text{otherwise} \end{cases} \\ O_L[\![\varnothing]\!]_c & = \{\} \\ O_R[\![.]\!] : \mathbf{Vrel} \rightarrow \mathbf{C} \rightarrow \mathbf{R} & \\ O_R[\![[R : vl]]\!]_c & = R(O_L[\![vl]\!]_c) \\ O_S[\![.]\!] : \mathbf{Vsch} \rightarrow \mathbf{C} \rightarrow \mathbf{S} & \\ O_S[\![\widehat{f}\langle \{VR_1; \ldots; VR_n\}, \varnothing \rangle]\!]_c = & \begin{cases} \{O_R[\![VR_1]\!]_c; \ldots; O_R[\![VR_n]\!]_c\}, & \text{if } F[\![\widehat{f}]\!]_c = \texttt{true} \\ \{\}, & \text{otherwise} \end{cases} \end{array}$$

Figure 5.1: Relational model and v-schema definitions.

formula of a variational set can only contain a subset of SPL features. For now, we do not simplify the formulas even if possible.

Example 5.1.7 *Assume we have the domain relation deposit defined in Example 5.1.2 and the domain relation member(ID, firstName, middleName, lastName). We also have feature categories type and country, where type may take feature booleans b, s, and i and country may take US and Ir. Remember that the middleName attribute is only used when US =* `true`*. Now assume that the SPL contains the variations of US with all three software types. However, it only includes the variation of banking in Iran. The v-schema in this situation is:*

$$\widehat{f}\langle\{[deposit : vl_{deposit}]\,;[memeber : vl_{member}]\}, \varnothing\rangle$$

where:

$$\begin{aligned}\widehat{f} &= ((US \wedge \neg Ir) \wedge (b \wedge \neg s \wedge \neg i)) \vee ((US \wedge \neg Ir) \wedge (\neg b \wedge s \wedge \neg i)) \\ &\quad \vee ((US \wedge \neg Ir) \wedge (\neg b \wedge \neg s \wedge i)) \vee ((\neg US \wedge Ir) \wedge (b \wedge \neg s \wedge \neg i)) \\ vl_{member} &= ((US \wedge \neg Ir)\,\langle\{middleName\}, \varnothing\rangle) \cup \{ID, firstName, lastName\}\end{aligned}$$

We have defined $vl_{deposit}$ in previous examples.

For reference, the abstract syntax and semantics of relational model is given in Figure 5.1. A schema is defined as a set of relation specifications (we call them specifications), where each specification provides the name of the relation and its attributes. Conceptually, a variational schema represents many different plain schemas that can be obtained by configurations.

5.2 Variational Object Equivalence

In this section, we establish syntactic rules for deriving the semantic equivalence of variational objects, including v-schema, variational relation, and variational set.

Definition 5.2.1 (Variational object equivalence) *Let ($\equiv$) be a binary relation on objects defined by $o \equiv o'$ iff $\forall c \in \mathbf{C} : O[\![o]\!]_c = O[\![o']\!]_c$.*

Note that the object equivalence relation definition ranges over all the semantic functions O_L, O_R, and O_S. In other words, it can be applied to all variational objects: attribute set, relation, and schema. One can easily specify the object type by determining it as a subscript, i.e. O_L, O_R, and O_S for attribute set, relation, and schema, respectively. With a similar reasoning to the feature formula equivalence relation, the relation ($\equiv$) is an equivalence relation, and objects o and o' are (semantically) equivalent if $o \equiv o'$.

5.3 The Representation System for a V-DB: V-Table

Now that we have introduced an abstraction to encode all valid schemas of the underlying domain of interest, the v-schema, we need a representation system for the v-DB to communicate with it as well as enabling the user to visualize their result to a query. To be able to represent a v-DB we have to represent its elements, i.e., the tuples in variational relations. It is essential to use a representation system that can effectively show the variability of data in tuple-level. Note that the v-schema takes care of the variability in the structure of data. Hence, we do not need to worry about structure variability while defining the representation system. We introduce our representation formalism, *variational table (v-table)*, as follows:

Definition 5.3.1 (Variational table) *A* **v-table***, short for* **variational table***, is a pair, $VT = (T, \Psi)$, defined over the variational database instance I, where T is a relation instance in I and $\Psi : \mathbf{R} \to \mathbf{F}$ assigns a feature formula, $\Psi(t)$, to tuple t to indicate the software variation(s) that it belongs to. We call the Ψ function the "variation presence condition assigner"* **(VPCA)***.*

Remark 5.3.2 *Note that a variational relation is an abstraction that encodes all possible relations of a domain relation. As opposed to a v-table that represents an instance of a variational relation in a v-DB instance. Hence, they perform different tasks. While the former manages the data variability, the latter one handles the structure variability.*

We call a feature formula that determines whether an element is valid in a given configuration *presence condition.* Therefore, we define relation presence conditions, attribute presence conditions, and tuple presence conditions. While the v-relation takes care of the first two, the v-table keeps track of the last one.

	ID	*firstName*	*middleName*	*lastName*
F_1	1	Sean	John	Patrick
	2	Caitlin	Elyse	Brennan
F_2	3	Mani	$\perp$	Tehrani
	4	Saeid	$\perp$	Ghafouri

Table 5.1: A v-table of the variational relation *member*. The $\perp$ denotes the case where an attribute, A, is not valid for a given tuple, t, i.e., the variation that t belongs to does not include attribute A.

Example 5.3.3 *The v-table of the member variational relation in Example 5.1.7 is shown in Table 5.1, where* $part_1$ *includes the first two tuples and* $part_2$ *includes the last two tuples. Therefore:*

$$\forall t \in part_1 : \Psi(t) = F_1 = (US \wedge \neg Ir) \wedge (\neg b \wedge \neg s \wedge i)$$
$$\forall t \in part_2 : \Psi(t) = F_2 = (\neg US \wedge Ir) \wedge (b \wedge \neg s \wedge \neg i)$$

Note that we infer presence conditions for v-relations and attributes using the defined v-shcema. Here the presence condition of the v-table $(member, \Psi)$*, denoted by* $\Phi(member)$ *is:*

$$\begin{aligned}\Phi(member) = &((US \wedge \neg Ir) \wedge (b \wedge \neg s \wedge \neg i)) \vee ((US \wedge \neg Ir) \wedge (\neg b \wedge s \wedge \neg i)) \\ &\vee ((US \wedge \neg Ir) \wedge (\neg b \wedge \neg s \wedge i)) \vee ((\neg US \wedge Ir) \wedge (b \wedge \neg s \wedge \neg i)).\end{aligned}$$

In a similar manner, we infer the presence conditions of attributes. For attribute A *we denote its presence condition by* $\Phi(A)$*. Hence, in this example we have:*

$$\begin{aligned}\Phi(ID) &= f_1 = \texttt{true} \\ \Phi(firstName) &= f_2 = \texttt{true} \\ \Phi(middleName) &= f_3 = US \wedge \neg Ir \\ \Phi(lastName) &= f_4 = \texttt{true}\end{aligned}$$

Given a v-table (T, Ψ), an *instantiation* of it w.r.t. configuration c is a v-table (R^c, Ψ) such that:
$T^c = \{\langle t(A_1), \ldots t(A_k)\rangle | t \in T$ *and* $F[\![\Phi(T)]\!]_c = \texttt{true}$ *and* $\forall 1 \leq i \leq k : F[\![\Phi(A_i)]\!]_c =$

	ID	*firstName*	*lastName*
c	3	Mani	Tehrani
c	4	Saeid	Ghafouri

Table 5.2: The relation of the instantiation of the v-table $(member, \Psi)$ given in Table 5.1.

$\texttt{true}$ *and* $F[\![\Psi(t)]\!]_c = \texttt{true}\}$. Note that $\Phi(T)$ and $\Phi(A_i)$ are taken from the v-relation definition correspondent to T.

For example, Table 5.2 shows the instantiation $(member^c, \Psi)$ of the v-table shown in Table 5.1 where $c = ((\neg US \wedge Ir) \wedge (b \wedge \neg s \wedge \neg i))$.

Remark 5.3.4 *Note that the v-table instantiation itself is a v-table. Also an empty set is considered as a v-table that its table is empty and as a result its VPCA function does not have any tuple to assign a presence condition to.*

5.4 Property of the Representation System, V-Table

Definition 5.4.1 (closeness) *A representation system for a v-DB is* **closed** *under a query language if the result of a query can be represented in that representation system as well as its input.*

Consider a representation formalism, F, such as the one introduced in Figure 6.2. Let D be a variational database represented in F. Given a query $Q \in L$, where L is a query language, if we can represent $Q(D)$ in F then F is closed under L. Clearly, any complete representation system is closed too, which gives us Corollary 5.4.2.

Corollary 5.4.2 *V-table is closed under V-SPJ algebra.*

Proof Assume we have a query Q that only contains V-SPJ operations over a v-table in a variational database instance I. Assume $VT = Q(I)$ is the result of running Q over I. To prove closeness of v-table under V-SPJ algebra, We need to show that VT is a v-table as well. Hence we need to construct all the elements of a v-table, i.e., its relation instance and VPCA functions. Let $VT = (T, \Psi)$. We need to define T and Ψ. We have already comprehensively explored this while defining the validation and dynamic semantics of V-SPJ in Section 6.3.2. And as it can be seen the result of a v-query is always a v-table.

The property of a representation formalism being closed under a query language is also known as *strong representation system* [2]. As a result, the variational table is a strong representation system under variational queries.

Chapter 6: Query Language

A user's least expectation of a database system is the ability to satisfy their information need, such as manipulating data and extracting information. A query language is a tool for such an interaction with a database. In this section, we introduce the variational query language, v-query for short, as a query language for a v-DB. We pick a subset of relational algebra operations known as *SPJ algebra* and adopt them in such a way that they will be applicable to a v-DB instance. We first explain the SPJ algebra briefly in Section 6.1 and then introduce v-queries in Section 6.2. We then explore different semantics of the introduced query language: the configuration selection and dynamic semantics in Sections 6.3.1 and 6.3.2, respectively. Finally we elaborate on the equivalence of v-query terms in Section 6.4.

6.1 SPJ Algebra Semantics

The elements, syntax, and semantics of SPJ algebra are shown in Figure 6.1. Note that we just use the relation names to write a query. However, the query will be run on an instance of the relation provide by the database instance. We define the *expression equivalence relation* ($\equiv$) as a binary relation on SPJ expressions defined by $e \equiv e'$ iff $E[\![e]\!] = E[\![e']\!]$. Hence, expressions e and e' are (semantically) equivalent if $e \equiv e'$. The SPJ algebra contains four operations: projection, selection, join, and Cartesian product. We explain the semantics of the operations, shown in Figure 6.1, briefly [2]:

Selection filters data horizontally, i.e., it returns a subset of tuples belonging to a table T based on a condition θ. We define it formally in Figure 6.1. Note that $\theta(t) : \mathbf{T} \rightarrow \mathbf{T}$ is the condition θ applied to the tuple $t \in T$ and returns a boolean value. Note that a query using this operation must follow the *attribute inclusion* and *value-attribute domain matching* constraints explained in Section 6.1.1.

Projection filters data vertically, i.e., it returns a set of attributes with their correspondent values in tuples. The formal definition is shown in Figure 6.1, where $t(A_i)$ represents the value of a tuple at attribute A_i. We use the notation of $\langle\!\langle t(A_1), \ldots, t(A_j) \rangle\!\rangle$

SPJ algebra generic objects:

$$\begin{array}{rcll}
k \in \mathbf{K} & & & \textit{Constant} \\
\bullet \in \mathbf{Op} & ::= & \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{!=} & \textit{Operation} \\
\theta \in \mathbf{Cond} & ::= & b \mid A \bullet k \mid A \bullet A \mid \neg\theta \mid \theta \vee \theta \mid \theta \wedge \theta & \textit{Condition} \\
\kappa \in \mathbf{Jcond} & ::= & b \mid A = A \mid \kappa \vee \kappa \mid \kappa \wedge \kappa & \textit{Join Condition}
\end{array}$$

SPJ algebra syntax:

$$\begin{array}{rcll}
e \in \mathbf{Exp} & ::= & T & \textit{Relation specification} \\
& \mid & \sigma_\theta e & \textit{Selection} \\
& \mid & \pi_l e & \textit{Projection} \\
& \mid & e \times e & \textit{Cartesian product} \\
& \mid & e \bowtie_\kappa e & \textit{Join}
\end{array}$$

Semantics of SPJ algebra:

$$\begin{aligned}
E[\![.]\!] &: \mathbf{Exp} \rightarrow \mathbf{T} \\
E[\![T]\!] &= T \\
E[\![\sigma_\theta e]\!] &= \{t \in E[\![e]\!] \mid \theta(t) = \texttt{true}, \mathit{attr}(\theta) \in \mathit{attr}(e)\} \\
E[\![\pi_l e]\!] &= \{\langle\!\langle t(A_1), \ldots, t(A_j)\rangle\!\rangle \mid t \in E[\![e]\!], l = A_1, \cdots, A_j\} \\
E[\![e_1 \times e_2]\!] &= \{\langle\!\langle t_1(1), \ldots t_1(k), t_2(1), \ldots, t_2(j)\rangle\!\rangle \mid t_1 \in E[\![e_1]\!], t_2 \in E[\![e_2]\!], \\
&\qquad \mathit{arity}(e_1) = k, \mathit{arity}(e_2) = j\} \\
E[\![e_1 \bowtie_\kappa e_2]\!]_I &= \{t \in E[\![\sigma_\kappa(e_1 \times e_2)]\!]\}
\end{aligned}$$

Attribute inclusion constraint:

$$\begin{array}{ll}
Q = \pi_l e & \forall A \in l : A \in \mathit{attr}(e) \\
Q = \sigma_\theta e & \forall A \in \mathit{attr}(\theta) : A \in \mathit{attr}(e)
\end{array}$$

Value-attribute domain matching constraint:

$$Q = \sigma_\theta e \quad \forall A \in \mathit{attr}(\theta) : \mathit{val}_A(\theta) \in \mathit{dom}(A)$$

Attribute domains matching constraint:

$$Q = e_1 \bowtie_\kappa e_2 \quad \forall A \in \mathit{attr}(\theta) : \mathit{val}_A(\theta) \in \mathit{dom}(A)$$

Figure 6.1: The SPJ algebra definitions.

to represent a tuple with the attribute set $\{A_1, \ldots A_j\}$. Note that a query using this operation must follow the *attribute inclusion constraint* explained in Section 6.1.1.

Cartesian product joins every tuple in table e_1 with every tuple in e_2, without applying any conditions to the returned tuples. In its definition, shown in Figure 6.1, $t_1(i), 1 \leq i \leq k$ and $t_2(i), 1 \leq i \leq j$ denote the value of the tuples t_1 and t_2 at attribute i^{th} of relations e_1 and e_2 respectively.

Join is used to relate two tables e_1 and e_2 based on a condition κ. As it can be seen in Figure 6.1 this operation can easily be defined using the combination of selection and Cartesian product. Hence, we do not introduce it here. Note that the conditions used in the join operation must be of the **Jcond** syntactic category. Also note that a query using this operation must follow the *attribute domains matching constraint* explained in Section 6.1.1.

Note that in all SPJ operations, the input can be the result of some other SPJ algebra expression, since the relational model is closed under relational algebra, which the SPJ algebra is a subset of [2]. In that case each expression will be computed, and the result will be passed to the next operation, as shown in the semantics of SPJ algebra in Figure 6.1.

6.1.1 SPJ Query Constraints

Any query written in the SPJ algebra must satisfy three constraints, if applicable. These constraints are stated in Figure 6.1. If a query does not follow these constraints, applying it to a database instance will result in a type error.

Attribute inclusion constraint ensures that attributes used in the attribute operand of a query are included in the relation's argument of it. For example, the attributes used in the set of projection operator must be attributes of the relation that the projection has been applied to.

Value-attribute domain matching constraint ensures that the value assigned to attributes in the conditional operands of a query matches the domain of that attribute in the relation the query is applied to. Note that we denote such a value by $val_A(\theta)$. For example, we cannot have a condition like ID = *'John'*, where $dom(ID) = \mathbb{N}$. Because $John \notin \mathbb{N}$.

Attribute domains matching constraint ensures that the equality of attributes in a

V-SPJ objects:

$$\begin{array}{rcl} VT \in \mathbf{Vtab} & ::= & (T, \Psi) \\ v\theta \in \mathbf{Vcond} & ::= & b \mid A \bullet k \mid A \bullet A \mid \neg v\theta \mid v\theta \vee v\theta \mid v\theta \wedge v\theta \mid \widehat{f}\langle v\theta, v\theta \rangle \end{array}$$

V-SPJ algebra syntax:

$$\begin{array}{rcll} ve \in \mathbf{Vexp} & ::= & VT & \textit{Variational Table} \\ & \mid & \sigma_{v\theta}\ ve & \textit{Variational Selection} \\ & \mid & \pi_{vl}\ ve & \textit{Variational Projection} \\ & \mid & ve \times ve & \textit{Variational Cartesian Product} \\ & \mid & \widehat{f}\langle ve, ve \rangle & \textit{Variational Expression Choices} \end{array}$$

Figure 6.2: V-SPJ algebra definitions.

conditional operand of a query is valid, i.e., the domain of attributes on the right-hand and left-hand sides of the condition is the same. This constraint, for example, prevents cases that an attribute of the integer domain is checked for equality to an attribute of the string domain. In other words, it prevents type errors.

6.2 Variational Query (V-Query)

Conceptually, a variational query describes a query that can be executed over any database instance consistent with the variational schema. The property that must hold between a variational query VQ and a variational schema VS is that for every plain query Q_c obtained from VQ by configuring with a function $c : \mathbf{F} \to \mathbf{T}$, Q_c is consistent with the corresponding plain schema S_c obtained by $O_S[\![VS]\!]_c$ with the same function c. That is, every variant query matches the corresponding variant schema. In Section 6.5, we prove that our query language, v-query, can encode this idea and also recover any of the conceptually potential results for any instance of the variational database.

The representation of variational queries in Figure 6.2 builds on the representation of plain queries in Figure 6.1. First, relations are generalized to accept a variational set of attributes with assigned feature formulas defined in the variational schema. Second, the variational conditions are annotated by feature formulas indicating in which variants the conditions are applicable. Third, each query is itself annotated by a feature formula indicating in which variants the query should be employed. In what follows we demonstrate the potential of v-queries and provide some examples for a better understanding of them.

Remark 6.2.1 *Similar to SPJ queries, we define V-SPJ queries using just the name of the v-relations. Although, note that the v-table associated to that name in the v-DB instance will be passed to the query.*

Remark 6.2.2 *The defined syntax of v-query allows for applying plain relational operations on a v-table. Note that the presence condition of relation, attributes, and tuples will determine the validity of the returned attributes and tuples. Example 6.2.3 demonstrates this point. Note that in our examples, we drop the VPCA function for simplicity. However, while passing a query to the database, this function will be passed too.*

Example 6.2.3 *Assume we have the v-table given in Table 5.1 and we want to get members' IDs and first names. The v-query $q_1 = \pi_{ID,firstName}(member)$ will accomplish this task. Obviously, the result is a v-table that contains first three columns of the Table 5.1, with the same VPCA function. If we wanted to get the middle names as well we could still use the simple v-query $q_2 = \pi_{ID,firstName,middleName}(member)$. The result this time contains the first four columns of the table with the same VPCA function. Although a more optimized case is to eliminate the cases where the middle name attribute is not defined and update the functions accordingly. We will consider such optimizations in our future work. Note that getting unoptimized data is specifically problematic when a user wants to query the results of q_2. She normally expects to be querying the case where she has middle names, i.e., she has focused on the matter of US database. Assume the result of q_2 is (T, Ψ). Now she wants to get the first names used in the US. She expects just to project the first name attribute of T since she is assuming that she has already filtered out the cases that do not have the middle name, in our example tuples associated with Iran. Hence, she runs the query $q_3 = \pi_{firstName}(T)$. Surprisingly, she gets first names of both countries, and hence she will not be satisfied. She could have avoid this problem by rewriting q_2 as $q_2' = \pi_{(US,\neg Ir)\langle(ID,firstName,middleName),\varnothing\rangle}(member)$. She can then easily run q_3 on the result of q_2' and get first names used in the US. This also demonstrates the necessity of including the feature specifications in query operands.*

firstName
Sean
Caitlin
Mani
Saeid

Table 6.1: The result of query q_1 over *member* v-table in Example 6.2.4, demonstrating the need of including the VPCA function in v-tables.

It is vital to pass feature specifications in query operators and also in the v-table and the v-relation, all as feature formulas, to be able to match the correct variations either structurally or contextually. Example 6.2.4 and 6.2.3 demonstrate the necessity of this issue.

Example 6.2.4 *We first show the necessity of defining the VPCA function for a v-table. Assume we did not have the* Ψ *function of the member v-table shown in Table 5.1. Evaluating the v-query* $q_1 = \pi_{\mathit{firstName}}(\mathit{member})$ *results in Table 6.1. As it can be seen, we have no idea which variation the member with name Sean belongs to. Hence we have lost the provenance, or originality, of the data. In Section 5.4 we prove that defining the VPCA function made the v-table representation a variation preserving representation system. In contexts that users deal with data from a variety of sources, this is a crucial issue to handle.*

While writing a v-query, one must carefully choose presence conditions used for the query operands. Some subtle details and points must be taken into account when writing the queries. Example 6.2.5 demonstrates a simple case. For now, we assume any given v-query is valid and correct semantically. Although, we plan to check the validity of a v-query in our future work.

Example 6.2.5 *Assume the country category feature includes three features US, Ir, and Ic; standing for the United States, Iran, and Iceland, respectively. These features represent different countries that have requested a software from the SPL company. A user wants to get the middle name when the US is set and the first name when the Ic is set. At first, she writes the v-query:*
$Q_1 = \pi_{(US \wedge \neg Ic \wedge \neg Ir)\langle(\mathit{middleName}),(\mathit{firstName})\rangle}(\mathit{member})$.

However, she notices that in the result returned she also has first names of the country Ir. Then she realizes that the feature formula she used does not convey her intent. Because after evaluation to `false` *it considers both countries, Iceland and Iran. For her v-query to return her desired set of tuples, she needs to modify her v-query to* $Q_2 = \pi_{(US \wedge \neg Ic \wedge \neg Ir)\langle(middleName),(\neg US \wedge Ic \wedge \neg Ir)\langle(firstName),\varnothing\rangle\rangle}(member)$.

6.3 V-SPJ Semantics

We introduce three semantic functions over the introduced V-SPJ algebra with regards to the nature of variational databases: 1) *Configuration selection semantic* definitions, selection semantics definitions for short, specify a v-query to a given specific configuration. 2) *Static semantic* rules ensure that a query written over a v-DB is well-formed and consistent with the v-schema. As mentioned earlier, we plan to write these rules in future and here we assume any written v-query is statically checked. 3) *Dynamic semantic* definitions apply a query to a v-DB instant and return the result in the form of a v-table. We dive into details regarding these semantics in Sections 6.3.1 and 6.3.2, respectively. The input to a v-query is assumed to be a v-table in all semantic definitions.

6.3.1 V-SPJ Configuration Selection Semantics

The selection semantics of a v-query Q takes a specific configuration, c, and construct a v-query of Q with plain arguments over an instantiation of the input v-table with respect to c. In other words, the selection semantics limits a query to only one variation of data chosen by the user. Figure 6.3 defines all selection semantic definitions. Note that we have extended the "equivalence relation" definition to variational queries and conditions as explained in Section 6.4.

Example 6.3.1 *Assume we have the variational query* Q_2 *given in Example 6.2.5. We want to get the plain query of* Q_2 *associated to the configuration* $c_1 = (US \wedge \neg Ic \wedge \neg Ir) \wedge (b \wedge \neg i \wedge \neg s)$. *We will have:*

$$E_E[\![Q_2]\!]_{c_1} = \pi_{middleName}(member^{c_1})$$

Remark 6.3.2 *Note that the selection semantics does not output a plain query in SPJ*

Configuration selection semantics of variational conditions:

$$
\begin{aligned}
E_C[\![.]\!]_c &: \mathbf{Vcond} \rightarrow \mathbf{C} \rightarrow \mathbf{Cond} \\
E_C[\![b]\!]_c &= b \\
E_C[\![A \bullet k]\!]_c &= \begin{cases} A \bullet k, \text{ if } F[\![\Phi(A)]\!]_c = \texttt{true} \\ \texttt{false}, \text{ otherwise} \end{cases} \qquad \textit{where } A \in \textit{attr}(R), (R, \Psi) \in \mathbf{Vtab} \\
E_C[\![A_1 \bullet A_2]\!]_c &= \begin{cases} A_1 \bullet A_2, \text{ if } F[\![\Phi(A_1)]\!]_c = F[\![\Phi(A_1)]\!]_c = \texttt{true} \\ \texttt{false}, \text{ otherwise} \end{cases} \\
&\textit{where } A \in \textit{attr}(R), (R, \Psi) \in \mathbf{Vtab} \\
E_C[\![\neg v\theta]\!]_c &= \neg E_C[\![v\theta]\!]_c \\
E_C[\![v\theta_1 \vee v\theta_2]\!]_c &= E_C[\![v\theta_1]\!]_c \vee E_C[\![v\theta_2]\!]_c \\
E_C[\![v\theta_1 \wedge v\theta_2]\!]_c &= E_C[\![v\theta_1]\!]_c \wedge E_C[\![v\theta_2]\!]_c \\
E_C[\![\widehat{f}\langle v\theta_1, v\theta_2\rangle]\!]_c &= \begin{cases} E_C[\![v\theta_1]\!]_c, \text{ if } F[\![\widehat{f}]\!]_c = \texttt{true} \\ E_C[\![v\theta_2]\!]_c, \text{ otherwise} \end{cases}
\end{aligned}
$$

Configuration selection semantics of variational expressions (queries):

$$
\begin{aligned}
E_E[\![.]\!]_c &: \mathbf{Vexp} \rightarrow \mathbf{C} \rightarrow \mathbf{Vexp} \\
E_E[\![VT]\!]_c &= VT^c \\
E_E[\![\sigma_{v\theta}\ ve]\!]_c &= \sigma_{E_C[\![v\theta]\!]_c}\ E_E[\![ve]\!]_c \\
E_E[\![\pi_{vl}\ ve]\!]_c &= \pi_{O_L[\![vl]\!]_c}\ E_E[\![ve]\!]_c \\
E_E[\![ve_1 \times ve_2]\!]_c &= E_E[\![ve_1]\!]_c \times E_E[\![ve_2]\!]_c \\
E_E[\![\widehat{f}\langle ve_1, ve_2\rangle]\!]_c &= \begin{cases} E_E[\![ve_1]\!]_c, \text{ if } F[\![\widehat{f}]\!]_c = \texttt{true} \\ E_E[\![ve_2]\!]_c, \text{ otherwise} \end{cases}
\end{aligned}
$$

Figure 6.3: Configuration selection semantics of V-SPJ algebra.

algebra since the input is an instantiation of v-table and as defined in Section 5.3 an instantiation of a v-table is a variational table where all tuples and attributes presence conditions assign them to one specific configuration.

6.3.2 V-SPJ Dynamic Semantics

The dynamic semantics of a v-query defines the evaluation of that query against an instance of a v-DB. Consider we have a variational database instance I and a v-query Q. Assume that the result of running Q over D, denoted by Q_I, is the v-table (T, Ψ). We need to figure out different pieces of this v-table, namely the tuples to return and the VPCA function, Ψ. Note that the instance of the v-table is encoded inside its definition implicitly and the v-query takes a v-table as input. Hence, when evaluating a v-query, we do not need to explicitly state the database instance since we are already passing it to the v-query by means of v-table.

Remark 6.3.3 *Note that the we do not define dynamic semantics for the third type of v-queries since due to the assumption that we have distinct v-table and attribute name, they will never occur. In future work, we will relax this assumption and provide the semantics for its cases.*

Remark 6.3.4 *We assume the compatibility of the feature formulas and presence conditions has already been checked by the static semantic rules. We demonstrate this by stating "$\forall c \in \mathbf{C}_{checked}$" in our dynamic rules definitions.*

We go over each of the operations separately and define its dynamic semantics:

Selection: The dynamic semantics of v-queries employing the selection operation is shown in Figures 6.4 to 6.5. Notice that a dynamic semantic definition takes a variational expression, applies it to an instance of the v-DB, and outputs a v-table. Note that if the condition operand of a selection query is not provided, we will assume that `true` has been passed as the argument and hence the given v-table will be returned to the user.

We provide a brief justification of these definitions: Since the selection operation does not filter any attributes, the output v-table will have all the attributes of the input v-table. We also want to keep track of the variation which each tuple belongs to. Hence the VPCA function may be more restricted by the conditions used in the v-query. However,

$D[\![.]\!] : \mathbf{Vexp} \rightarrow \mathbf{Vtab}$

Dynamic semantics of variational selection queries (especial cases):

$D[\![(T, \Psi)]\!] = (T, \Psi)$

$D[\![\sigma_{\texttt{true}}\,(T, \Psi)]\!] = (T, \Psi)$

$D[\![\sigma_{\texttt{false}}\,(T, \Psi)]\!] = \{\}$

$$D[\![\sigma_{\texttt{true}}\left(\widehat{f}_t\langle(T_1, \Psi_1), (T_2, \Psi_2)\rangle\right)]\!] = (T, \Psi)$$

$$(\neq D[\![(T_1, \Psi_1) \times (T_2, \Psi_2)]\!])$$

$$\begin{aligned}
&\textit{where: } attr(T) = \{A \ \mid\ A \in attr(T_1) \cup attr(T_2)\}\\
&\qquad\qquad \textit{s.t. } arity(T) = arity(T_1) + arity(T_2)\\
&\qquad T = T_1' \cup T_2'\\
&\qquad\quad \textit{where: } T_1' = \{\langle\!\langle t(1), \ldots, t(j), \bot, \ldots, \bot\rangle\!\rangle \ \mid\ t \in T_1, arity(T_1) = j\}\\
&\qquad\qquad\quad T_2' = \{\langle\!\langle \bot, \ldots, \bot, t(1), \ldots, t(k)\rangle\!\rangle \ \mid\ t \in T_2, arity(T_2) = k\}\\
&\qquad\qquad \textit{s.t. } arity(T_1') = arity(T_2')\\
&\qquad\qquad\quad \forall t \in T : \Psi(t) = \begin{cases} \Psi_1\,(part\,(t, T_1)) \wedge \widehat{f}_t, & t \in T_1' \\ \Psi_2\,(part\,(t, T_2)) \wedge \neg\widehat{f}_t, & t \in T_2' \end{cases}
\end{aligned}$$

Dynamic semantics of type 1 variational selection query:

$D[\![\sigma_\theta\,(T_1, \Psi_1)]\!] = (T, \Psi)$

$$\begin{aligned}
&\textit{where: } attr(T) = attr(T_1)\\
&\qquad T = \{t \ \mid\ t \in T_1, \theta(t) = \texttt{true}\}\\
&\qquad \textit{s.t. } \forall t \in T : \Psi(t) = \Psi_1(t)
\end{aligned}$$

Figure 6.4: Dynamic semantics of variational queries using the selection operator. The *part* function takes a tuple t and a set of tuples T and returns the part of t that belongs to T. This function helps tracking what the variation of a constructed tuple is, i.e., whether a tuple t comes from T_1 or T_2. In other words, it drops the empty values of a tuple constructed in our definition.

Dynamic semantics of type 2 variational selection query:

$$
\begin{aligned}
&D[\![\sigma_{\widehat{f_\theta}\langle v\theta_1, v\theta_2\rangle}(T_1, \Psi_1)]\!] = (T, \Psi)\\
&\qquad \textit{where: } \mathit{attr}(T) = \mathit{attr}(T_1)\\
&\qquad\qquad T = T_1' \cup T_2'\\
&\qquad\qquad \textit{where: } T_1' = \{t \mid t \in T_1, (\forall c \in \mathbf{C}_{\mathit{checked}} : E_C[\![v\theta_1]\!]_c(t) = \texttt{true}, F[\![\Psi_1(t)]\!]_c = \texttt{true})\}\\
&\qquad\qquad\qquad T_2' = \{t \mid t \in T_2, (\forall c \in \mathbf{C}_{\mathit{checked}} : E_C[\![v\theta_2]\!]_c(t) = \texttt{true}, F[\![\Psi_1(t)]\!]_c = \texttt{true})\}\\
&\qquad\qquad s.t.\ \forall t \in T : \Psi(t) = \begin{cases} \Psi_1(t) \wedge \widehat{f_\theta}, & t \in T_1' \\ \Psi_2(t) \wedge \neg\widehat{f_\theta}, & t \in T_2' \end{cases}
\end{aligned}
$$

Dynamic semantics of type 4 variational selection query:

$$
\begin{aligned}
&D[\![\sigma_{\widehat{f_\theta}\langle v\theta_1, v\theta_2\rangle}\left(\widehat{f_t}\langle (T_1, \Psi_1), (T_2, \Psi_2)\rangle\right)]\!] = (T, \Psi)\\
&\qquad \textit{where: } \mathit{attr}(T) = \{A \mid A \in \mathit{attr}(T_1) \cup \mathit{attr}(T_2)\}\\
&\qquad\qquad\qquad s.t.\ \mathit{arity}(T) = \mathit{arity}(T_1) + \mathit{arity}(T_2)\\
&\qquad\qquad T = T_1' \cup T_1'' \cup T_2' \cup T_2''\\
&\qquad\qquad \textit{where: } T_1' = \{\langle\!\langle t(1), \ldots, t(j), \bot, \ldots, \bot\rangle\!\rangle \mid t \in T_1, \mathit{arity}(T_1) = j,\\
&\qquad\qquad\qquad\qquad (\forall c \in \mathbf{C}_{\mathit{checked}} : E_C[\![v\theta_1]\!]_c(t) = \texttt{true}, F[\![\Psi_1(t)]\!]_c = \texttt{true})\}\\
&\qquad\qquad\qquad T_1'' = \{\langle\!\langle t(1), \ldots, t(j), \bot, \ldots, \bot\rangle\!\rangle \mid t \in T_1, \mathit{arity}(T_1) = j,\\
&\qquad\qquad\qquad\qquad (\forall c \in \mathbf{C}_{\mathit{checked}} : E_C[\![v\theta_2]\!]_c(t) = \texttt{true}, F[\![\Psi_1(t)]\!]_c = \texttt{true})\}\\
&\qquad\qquad\qquad T_2' = \{\langle\!\langle \bot, \ldots, \bot, t(1), \ldots, t(k)\rangle\!\rangle \mid t \in T_2, \mathit{arity}(T_2) = k,\\
&\qquad\qquad\qquad\qquad (\forall c \in \mathbf{C}_{\mathit{checked}} : E_C[\![v\theta_1]\!]_c(t) = \texttt{true}, F[\![\Psi_2(t)]\!]_c = \texttt{true})\}\\
&\qquad\qquad\qquad T_2'' = \{\langle\!\langle \bot, \ldots, \bot, t(1), \ldots, t(k)\rangle\!\rangle \mid t \in T_2, \mathit{arity}(T_2) = k,\\
&\qquad\qquad\qquad\qquad (\forall c \in \mathbf{C}_{\mathit{checked}} : E_C[\![v\theta_2]\!]_c(t) = \texttt{true}, F[\![\Psi_2(t)]\!]_c = \texttt{true})\}\\
&\qquad\qquad s.t.\ \forall t \in T : \Psi(t) = \begin{cases} \Psi_1(\mathit{part}(t, T_1)) \wedge \widehat{f_t} \wedge \widehat{f_\theta}, & t \in T_1' \\ \Psi_1(\mathit{part}(t, T_1)) \wedge \widehat{f_t} \wedge \neg\widehat{f_\theta}, & t \in T_1'' \\ \Psi_2(\mathit{part}(t, T_2)) \wedge \neg\widehat{f_t} \wedge \widehat{f_\theta}, & t \in T_2' \\ \Psi_2(\mathit{part}(t, T_2)) \wedge \neg\widehat{f_t} \wedge \neg\widehat{f_\theta}, & t \in T_2'' \end{cases}
\end{aligned}
$$

Figure 6.5: Dynamic semantics of variational queries using the selection operator (continued).

tuples will not lose their originality. For example, assume a tuple, t, belongs to variations c_1 and c_2. The v-query applies a condition that filters out tuples of variation c_2 from the output. Then the returned tuple correspondent to t will only be of variation c_1, which is more restrictive than before.

Projection: The dynamic semantic definitions of v-queries that use the projection operator are shown in Figures 6.6 and 6.7. The output relation will include the attribute set resulted from the dynamic definition of the attribute set operand in the query, i.e., $L[\![vl]\!]$ in $\pi_{vl}(VT)$. Again the VPCA function will be modified according to the v-query,

Cartesian product: The dynamic semantic definitions of a v-query using the Cartesian product operation are provided in Figure 6.8. Note that each resulting tuple consists of two parts: one part that belongs to the first v-table and a second portion that belongs to the second given v-table, i.e., a returning tuple is a concatenation of a tuple in the first and second v-tables. The VPCA function will change accordingly based on the v-query.

Remark 6.3.5 *Note that in all dynamic semantic definitions, the VPCA function is operating such that the originality of tuples is not misplaced, i.e., after applying a query the variation of a tuple is either the same as it was or it is being more restricted due to the conditions in the v-query. However, the information about what variation a tuple belongs to will not be lost or generalized. We call this property of variational databases* **variation preservation**. *It is essential to note that the VPCA function makes the variational table a variation preserving representation system.*

6.4 Variational Query Equivalence

In this section, we define and prove sound a syntactic relation ($\equiv$) on variational queries, where if two expression $ve_1 \equiv ve_2$ are related, then ve_1 and ve_2 are semantically equivalent, that is, $D[\![ve_1]\!] = D[\![ve_2]\!]$. Note that since every expression defined in the V-SPJ algebra is a query we use the terms *query* and *expression* interchangeably. In order to introduce this relation we first define the syntactic relation ($\equiv_c$) on variational queries (and conditions), where if two terms $vt_1 \equiv vt_2$ are related, then vt_1 and vt_2 are semantically equivalent w.r.t. configuration c, that is, $E[\![ve_1]\!]_c = E[\![ve_2]\!]_c$, where E can either be subscripted with C or E for variational conditions and expressions respectively. Note that we define the first equivalence relation both independent and with the help

Dynamic semantics of a variational attribute set:

$$\begin{aligned}
&L[\![.]\!] : \mathbf{Vset} \to \mathbf{L}\\
&L[\![A]\!] = \{A\}\\
&L[\![vl_1 \cup vl_2]\!] = \{A \mid \exists c \in \mathbf{C} : A \in O_L[\![vl_1 \cup vl_2]\!]_c\}\\
&L[\![\widehat{f}\langle vl_1, vl_2\rangle]\!] = \{A \mid \exists c \in \mathbf{C} : A \in O_L[\![\widehat{f}\langle vl_1, vl_2\rangle]\!]_c\}\\
&L[\![\varnothing]\!] = \{\}
\end{aligned}$$

Dynamic semantics of variational projection queries (especial cases):

$$\begin{aligned}
&D[\![\pi_{\varnothing}\,(T_1, \Psi_1)]\!] = \{\}\\
&D[\![\pi_{\varnothing}\left(\widehat{f}_t\langle (T_1, \Psi_1)\,, (T_2, \Psi_2)\rangle\right)]\!] = \{\}
\end{aligned}$$

Dynamic semantics of type 1 variational selection query:

$$\begin{aligned}
D[\![\pi_l\,(T_1, \Psi_1)]\!] &= (T, \Psi)\\
&\textit{where: } attr(T) = l\\
&\qquad T = \{\langle\!\langle t(A_1), \ldots, t(A_k)\rangle\!\rangle \mid t \in T_1, l = \{A_1, \ldots, A_k\}\}\\
&\qquad s.t.\ \forall t \in T : \Psi(t) = \Psi_1(t)
\end{aligned}$$

Dynamic semantics of type 2 variational selection query:

$$\begin{aligned}
D[\![\pi_{\widehat{f_l}\langle vl_1, vl_2\rangle}\,(T_1, \Psi_1)]\!] &= (T, \Psi)\\
&\textit{where: } attr(T) = L[\![\widehat{f_l}\langle vl_1, vl_2\rangle]\!]\\
&\qquad T = \{\langle\!\langle t(A_1), \ldots, t(A_k)\rangle\!\rangle \mid t \in T_1, L[\![\widehat{f_l}\langle vl_1, vl_2\rangle]\!] = \{A_1, \ldots, A_k\}\}\\
&\qquad s.t.\ \forall t \in T : \Psi(t) = \begin{cases} \Psi_1(t) \wedge \widehat{f_l}, & \forall A \in L[\![vl_2]\!] : t(A) = \perp\\ \Psi_2(t) \wedge \neg\widehat{f_l}, & \forall A \in L[\![vl_1]\!] : t(A) = \perp \end{cases}
\end{aligned}$$

Figure 6.6: Dynamic semantics of variational sets and variational queries with the projection operator.

Dynamic semantics of type 4 variational selection query:

$$
\begin{aligned}
&D[\![\pi_{\widehat{f_l}\langle vl_1, vl_2\rangle} \widehat{f_t}\langle (T_1, \Psi_1), (T_2, \Psi_2)\rangle]\!] = (T, \Psi)\\
&\qquad \textit{where: } \mathit{attr}(T) = L[\![\widehat{f_l}\langle vl_1, vl_2\rangle]\!]\\
&\qquad\qquad T = \{\langle\!\langle t(A_1), \ldots, t(A_k)\rangle\!\rangle | (t \in T_1 \ \textit{or}\ t \in T_2), L[\![\widehat{f_l}\langle vl_1, vl_2\rangle]\!] = \{A_1, \ldots, A_k\}\}\\
&\qquad\qquad \textit{s.t.}\ \forall t \in T : \Psi_t = \begin{cases} \Psi_1(t) \wedge \widehat{f_t} \wedge \widehat{f_l}, & t \in T_1, \forall A \in L[\![vl_2]\!] : t(A) = \perp \\ \Psi_1(t) \wedge \widehat{f_t} \wedge \neg\widehat{f_l}, & t \in T_1, \forall A \in L[\![vl_1]\!] : t(A) = \perp \\ \Psi_2(t) \wedge \neg\widehat{f_t} \wedge \widehat{f_l}, & t \in T_2, \forall A \in L[\![vl_2]\!] : t(A) = \perp \\ \Psi_2(t) \wedge \neg\widehat{f_t} \wedge \neg\widehat{f_l}, & t \in T_2, \forall A \in L[\![vl_1]\!] : t(A) = \perp \end{cases}
\end{aligned}
$$

Figure 6.7: Dynamic semantics of variational sets and variational queries with the projection operator (continued).

of the second equivalence relation definition. The rules that make up these equivalence relations can, therefore, be used to prove the semantic equivalence of two variational queries, or if applied in a directed fashion, can be used to transform a variational query in a semantics-preserving way.

Definition 6.4.1 (Variational query equivalence w.r.t. a configuration c) *Let $(\equiv_c$ $)$ be a binary relation w.r.t. c on variational queries defined by $ve \equiv_c ve'$ iff $E_E[\![ve]\!]_c = E_E[\![ve']\!]_c$.*

Definition 6.4.2 (Variational condition equivalence w.r.t. a configuration c) *Let $(\equiv_c)$ be a binary relation w.r.t. c on variational conditions defined by $v\theta \equiv_c v\theta'$ iff $E_C[\![v\theta]\!]_c = E_C[\![v\theta']\!]_c$.*

Definition 6.4.3 (Variational expression equivalence) *Let $(\equiv)$ be a binary relation on variational queries defined by 1) $ve \equiv ve'$ iff $\forall I \in \mathbf{I} : D[\![ve]\!] = D[\![ve']\!]$ or 2) $ve \equiv ve'$ iff $\forall c \in \mathbf{C}_{checked} : E_E[\![ve]\!]_c = E_E[\![ve']\!]_c$.*

The relational algebra equivalence rules focus on optimizations in order to run the most efficient expression at the runtime. Similarly, formula choice calculus (FCC) equivalence rules try to remove redundancy and unemployed branches, which would result in better performance as well [11]. The variational query equivalence rules combine any of these rules together to achieve a more optimized query with as little as possible

Dynamic semantics of type 1 variational Cartesian product query:

$$
\begin{aligned}
&D[\![(T_1, \Psi_1) \times (T_2, \Psi_2)]\!] = (T, \Psi)\\
&\qquad \textit{where: } attr(T) = attr(T_1) \cup attr(T_2)\\
&\qquad\qquad \textit{s.t. } arity(T) = arity(T_1) + arity(T_2)\\
&\qquad\qquad T = \{\langle\!\langle t_1(1), \ldots, t_1(j), t_2(1), \ldots, t_2(k)\rangle\!\rangle \mid t_1 \in T_1, t_2 \in T_2, arity(T_1) = j,\\
&\qquad\qquad\qquad arity(T_2) = k, \Psi_1(t_1) \wedge \Psi_2(t_2) \neq \texttt{false}\}\\
&\qquad\qquad \textit{s.t. } \forall t \in T, t_1 \in T_1, t_2 \in T_2, t = concat(t_1, t_2) : \Psi_t = \Psi_1(t_1) \wedge \Psi_2(t_2)
\end{aligned}
$$

Dynamic semantics of type 2 variational Cartesian product query:

$$
\begin{aligned}
&D[\![\widehat{f_t}\langle(T_1, \Psi_1), (T_1', \Psi_1')\rangle \times \widehat{f_{tt}}\langle(T_2, \Psi_2), (T_2', \Psi_2')\rangle]\!] = (T, \Psi)\\
&\quad \textit{where: } attr(T) = attr(T_1) \cup attr(T_1') \cup attr(T_2) \cup attr(T_2')\\
&\qquad\quad \textit{s.t. } arity(T) = arity(T_1) + arity(T_2) + arity(T_1') + arity(T_2')\\
&\qquad T = P_1 \cup P_2 \cup P_3 \cup P_4\\
&\qquad \textit{where: } P_1 = \{\langle\!\langle t_1(1), \ldots, t_1(j), t_2(1), \ldots, t_2(k), \bot, \ldots, \bot\rangle\!\rangle \mid t_1 \in T_1, t_2 \in T_2,\\
&\qquad\qquad\qquad arity(T_1) = j, arity(T_2) = k, \Psi_1(t_1) \wedge \Psi_2(t_2) \neq \texttt{false}\}\\
&\qquad\qquad P_2 = \{\langle\!\langle t_1(1), \ldots, t_1(j), \bot, \ldots, \bot, t_2(1), \ldots, t_2(n)\rangle\!\rangle \mid t_1 \in T_1, t_2 \in T_2',\\
&\qquad\qquad\qquad arity(T_1) = j, arity(T_2') = n, \Psi_1(t_1) \wedge \Psi_2'(t_2) \neq \texttt{false}\}\\
&\qquad\qquad P_3 = \{\langle\!\langle \bot, \ldots, \bot, t_1(1), \ldots, t_1(m), t_2(1), \ldots, t_2(k), \bot, \ldots, \bot\rangle\!\rangle \mid t_1 \in T_1',\\
&\qquad\qquad\qquad t_2 \in T_2, arity(T_1') = m, arity(T_2) = k, \Psi_1'(t_1) \wedge \Psi_2(t_2) \neq \texttt{false}\}\\
&\qquad\qquad P_4 = \{\langle\!\langle \bot, \ldots, \bot, t_1(1), \ldots, t_1(m), \bot, \ldots, \bot, t_2(1), \ldots, t_2(n)\rangle\!\rangle \mid t_1 \in T_1',\\
&\qquad\qquad\qquad t_2 \in T_2', arity(T_1') = m, arity(T_2') = n, \Psi_1'(t_1) \wedge \Psi_2'(t_2) \neq \texttt{false}\}\\
&\qquad\qquad \textit{s.t. } arity(P_1) = arity(P_2) = arity(P_3) = arity(P_4)\\
&\qquad\qquad\quad \forall t \in T : \Psi(t) = \begin{cases}
\Psi_1\left(part\left(t, T_1\right)\right) \wedge \Psi_2\left(part\left(t, T_2\right)\right) \wedge \widehat{f_t} \wedge \widehat{f_{tt}}, & t \in P_1\\
\Psi_1\left(part\left(t, T_1\right)\right) \wedge \Psi_2'\left(part\left(t, T_2'\right)\right) \wedge \widehat{f_t} \wedge \neg\widehat{f_{tt}}, & t \in P_2\\
\Psi_1'\left(part\left(t, T_1'\right)\right) \wedge \Psi_2\left(part\left(t, T_2\right)\right) \wedge \neg\widehat{f_t} \wedge \widehat{f_{tt}}, & t \in P_3\\
\Psi_1'\left(part\left(t, T_1'\right)\right) \wedge \Psi_2'\left(part\left(t, T_2'\right)\right) \wedge \neg\widehat{f_t} \wedge \neg\widehat{f_{tt}}, & t \in P_4
\end{cases}
\end{aligned}
$$

Figure 6.8: Dynamic semantics of variational queries with the Cartesian product operator. For brevity, we state: $concat(t_1, t_2) = \langle\!\langle t_1(1), \ldots, t_1(j), t_2(1), \ldots, t_2(k)\rangle\!\rangle$.

redundancy. Note that they should be combined accordingly in order to get the correct equivalence rule. These rules can be used to optimize the query trees by both simplifying the feature formulas and the relational algebra operations. We state and sound prove some of the interesting variational query equivalences. The most important equivalence rule is the choice distributive rule with other operations as shown in Figure 6.9. Figure 6.10 demonstrates some of the relational algebra and FCC equivalence rules.

Theorem 6.4.4 *The choice distributive rules (w.r.t. configuration c) hold for all feature formulas $\widehat{f} \in \mathcal{F}$, variational expressions $ve_1, \ldots, ve_4 \in$ **Vexp**, variational attribute lists $vl_1, vl_2 \in$ **Vset**, and variational conditions $v\theta_1, v\theta_2 \in$ **Vcond**; $v\kappa_1, v\kappa_2 \in$ **Vjcond** (and a given valid configuration c).*

Proof We prove the Projection-Choice distributive equivalence (w.r.t. c) directly from the definitions provided in Figure 6.3. The rest can be proved similarly.

$$\forall c \in \mathbf{C}_{checked} :$$

$$\begin{aligned}
E_E[\![\widehat{f}\langle \pi_{vl_1} ve_1, \pi_{vl_2} ve_2\rangle]\!]_c &= \begin{cases} E_E[\![\pi_{vl_1} ve_1]\!]_c, & \text{if } F[\![\widehat{f}]\!]_c = \texttt{true} \\ E_E[\![\pi_{vl_2} ve_2]\!]_c, & \text{otherwise} \end{cases} \\
&= \begin{cases} \pi_{O_L[\![vl_1]\!]_c} E_E[\![ve_1]\!]_c, & \text{if } F[\![\widehat{f}]\!]_c = \texttt{true} \\ \pi_{O_L[\![vl_2]\!]_c} E_E[\![ve_2]\!]_c, & \text{otherwise} \end{cases} \\
&= E_E[\![\pi_{f\langle vl_1, vl_2\rangle} \widehat{f}\langle ve_1, ve_2\rangle]\!]_c
\end{aligned}$$

6.5 Properties of the Query Language

As mentioned in Section 6.2 the nature of variational databases requires the variational query to represent any function $c : \mathbf{C} \to \mathbf{Exp}$. To demonstrate this capability of v-queries we define *total variational expressive* query languages and prove that the v-query is a total variational expressive language.

Definition 6.5.1 (Total variational expressiveness) *A query language L is a* **total variational expressive** *query language for variational databases if there exists a function $f : \mathbf{Exp} \to \mathbf{C} \to \mathbf{Vexp}$ and its reverse f^{-1} such that they can translate a query in L to a set of queries in L' in different variant configurations and vice versa.*

Distributive rules w.r.t. configuration c:

$$\widehat{f}\langle \pi_{vl_1} ve_1, \pi_{vl_2} ve_2\rangle \equiv_c \pi_{\widehat{f}\langle vl_1, vl_2\rangle} \widehat{f}\langle ve_1, ve_2\rangle \qquad \widehat{f}\langle \sigma_{v\theta_1} ve_1, \sigma_{v\theta_2} ve_2\rangle \equiv_c \sigma_{\widehat{f}\langle v\theta_1, v\theta_2\rangle} \widehat{f}\langle ve_1, ve_2\rangle$$

$$\widehat{f}\langle ve_1 \times ve_2, ve_3 \times ve_4\rangle \equiv_c \widehat{f}\langle ve_1, ve_3\rangle \times \widehat{f}\langle ve_2, ve_4\rangle$$

Distributive rules:

$$\widehat{f}\langle \pi_{vl_1} ve_1, \pi_{vl_2} ve_2\rangle \equiv \pi_{\widehat{f}\langle vl_1, vl_2\rangle} \widehat{f}\langle ve_1, ve_2\rangle \qquad \widehat{f}\langle \sigma_{v\theta_1} ve_1, \sigma_{v\theta_2} ve_2\rangle \equiv \sigma_{\widehat{f}\langle v\theta_1, v\theta_2\rangle} \widehat{f}\langle ve_1, ve_2\rangle$$

$$\widehat{f}\langle ve_1 \times ve_2, ve_3 \times ve_4\rangle \equiv \widehat{f}\langle ve_1, ve_3\rangle \times \widehat{f}\langle ve_2, ve_4\rangle$$

Figure 6.9: V-SPJ algebra and choice distributive rules.

$$\widehat{f}\langle \sigma_{v\theta_1 \wedge v\theta_2} ve_1, \sigma_{v\theta_1 \wedge v\theta_3} ve_2\rangle \equiv \sigma_{v\theta_1}(\widehat{f}\langle \sigma_{v\theta_2} ve_1, \sigma_{v\theta_3} ve_2\rangle) \equiv \sigma_{v\theta_1 s\langle v\theta_2, v\theta_3\rangle} \widehat{f}\langle ve_1, ve_2\rangle$$
$$\widehat{f}\langle \pi_{vl_1}(\pi_{vl_2} ve_1), \pi_{vl_1}(\pi_{vl_3} ve_2)\rangle \equiv \pi_{vl_1} \widehat{f}\langle ve_1, ve_2\rangle$$
$$\widehat{f}\langle \pi_{vl_1}(\pi_{vl_2} e_1), \pi_{vl_3}(\pi_{vl_2} ve_2)\rangle \equiv \widehat{f}\langle \pi_{vl_1} ve_1, \pi_{vl_3} e_2\rangle \equiv \pi_{\widehat{f}\langle vl_1, vl_3\rangle} \widehat{f}\langle ve_1, ve_2\rangle$$
$$\widehat{f}\langle \sigma_{v\theta_1}(\sigma_{v\theta_2} ve_1), \sigma_{v\theta_3}(\sigma_{v\theta_1} ve_2)\rangle \equiv \sigma_{v\theta_1}(\widehat{f}\langle \sigma_{v\theta_2} ve_1, \sigma_{v\theta_3} ve_2\rangle)$$

Figure 6.10: Combinations of relational algebra and FCC rules.

Theorem 6.5.2 *The v-query is a total variational expressive query language w.r.t. SPJ algebra.*

In order to prove Theorem 6.5.2 we break down the Definition 6.5.1 into two definitions and prove lemmas to prove Theorem 6.5.2 in parts.

Definition 6.5.3 (Type 1 expressiveness) *A query language L is a* **type 1 expressive** *language for variational databases with respect to query language L' if there exists a function $f(c, Q)$ that for a given configuration c translates a query $Q \in L$ to a query $Q' \in L'$, where L' is a query language for relational data.*

Lemma 6.5.4 *The v-query is a type 1 expressive language for variational databases w.r.t. SPJ algebra.*

The proof of this lemma is trivial since the function f is the V-SPJ configuration semantics defined in Section 6.3.1. V-SPJ algebra is more expressive than just expressing all queries possible in SPJ due to its ability to run multiple SPJ queries simultaneously over different variations. Remember that v-SPJ provides the opportunity of running all plain queries included compactly in a v-query over a set of variations simultaneously. We introduce the second type of expressiveness to indicate this property.

Definition 6.5.5 (Type 2 expressiveness) *A query language L is a* **type 2 expressive** *language over variational databases w.r.t. the query language L' if given any set of queries in L', there exists a query $Q \in L$ that captures all the elements of the given set in different variations.*

Lemma 6.5.6 *The v-query is a type 2 expressive language for variational databases w.r.t. SPJ algebra.*

Proof We prove this theorem by construction. Without loss of generality, assume we have plain queries $q_1, \ldots, q_n$ written in SPJ algebra over instantiations $T^{c_1}, \ldots, T^{c_n}$ of the v-table (T, Ψ), where $\forall c_i, c_j : c_i \wedge c_j = \texttt{false}; 1 \leq i \leq j \leq n$. Following the syntax and semantics of v-query introduced in Figure 6.2 we can construct the v-query Q to encode all $q_1, \ldots, q_n$:

$$Q = c_1\langle q_1, c_2\langle q_2, c_3\langle q_3, \cdots, c_n\langle q_n, \varnothing\rangle \cdots\rangle\rangle$$

It is clear that Q represents all $q_1, \ldots, q_n$. Hence, v-query is a type 2 expressive language w.r.t. SPJ algebra.

Chapter 7: Conclusion

In this thesis, we introduced principles required to implement a variational database system such as variational schema, variational query, and v-table. Such a system is capable of abstracting over structural differences of several related databases that can be represented using the introduced feature model. This property allows for variational schema and variational queries, which provides the user with the ability to query multiple databases simultaneously and get a set of results corresponding to the used databases. The flexibility and expressiveness of the variational query language relax the dependency of queries on the representation of the data. We focused on the application of software product lines. However, we plan to generalize this idea such that it can be employed in the domains mentioned in the introduction.

We plan to implement such a system on top of an RDBMS. In order for such a system to be effective, we need to introduce the translation of variational queries to plain queries in addition to the semantics functions presented in this work. In addition to that, we plan to employ the equivalence rules to optimize the query tree and thus achieve an efficient system. We also plan to make the system as much automatic as possible, e.g., the system will generate the variational schema instead of the user providing it. We introduced different methods of encoding in Chapter **??** and provided a comprehensive comparison among them and plan to test the theoretical comparison with experimental studies. It worth exploring whether building a variational database on top of an RDBMS is more efficient than implementing a variational database system from scratch. While we can take advantage of what traditional RDBMSs have to offer, implementing such a system from scratch may be more efficient provided a variational nature to it.

Bibliography

[1] *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level.* Addison-Wesley, 1994.

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

[4] Parisa Ataei, Arash Termehchy, and Eric Walkingshaw. Variational databases. In *Proceedings of The 16th International Symposium on Database Programming Languages, DBPL 2017, Munich, Germany, September 1, 2017*, pages 11:1–11:4, 2017.

[5] Carlo Curino, Hyun Moon, and Carlo Zaniolo. Graceful Database Schema Evolution: the PRISM Workbench. In *VLDB*, 2008.

[6] Nilesh Dalvi, Philip Bohannon, and Fei Sha. Robust web extraction: An approach based on a probabilistic tree-edit model. SIGMOD, pages 335–348, New York, NY, USA, 2009. ACM.

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

[8] 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.

[9] 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, 2013.

[10] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data Exchange: Semantics and Query Answering. In *ICDT*, 2003.

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

[12] 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.

[13] Niloofar Khedri and Ramtin Khosravi. Handling database schema variability in software product lines. In *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 1*, pages 331–338, 2013.

[14] Adi Omari, Sharon Shoham, and Eran Yahav. Synthesis of forgiving data extractors. WSDM '17, pages 385–394. ACM, 2017.

[15] Aditya Parameswaran, Nilesh Dalvi, Hector Garcia-Molina, and Rajeev Rastogi. Optimal schemes for robust web extraction. In *Proceedings of the VLDB Endowment*, Berlin, Heidelberg, 2011.

[16] Jose Picado, Arash Termehchy, Alan Fern, and Parisa Ataei. Schema independent relational learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 929–944, New York, NY, USA, 2017. ACM.

[17] Dan Suciu, Dan Olteanu, R. Christopher, and Christoph Koch. *Probabilistic Databases*. Morgan & Claypool Publishers, 1st edition, 2011.

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

[19] 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.