A Variational Database Management System

Parisa Ataei
ataeip@oregonstate.edu
Oregon State University

Fariba Khan
khanfari@oregonstate.edu
Oregon State University

Eric Walkingshaw
walkiner@oregonstate.edu
Oregon State University

Abstract

Many problems require working with data that varies in its structure and content. Likewise, many tools have been developed to deal with different kinds of this problem, e.g., schema evolution tools or data integration tools. However, these approaches address only the specific kind of variation in databases resulting in two problems: Some fail to address all users' needs for their specific kind of variation and all fail to address the interaction of different kinds of variation in databases. For example, neither schema evolution nor data integration systems can manage the data variation when an integrated database evolves over time. In this paper, we define a generic framework that explicitly accounts for variation in databases. Thus, it captures variation in the structure and content of a database and allows queries to express variational information needs. Our framework adds expressiveness to databases since it can encode any kind of variation. We design and implement a variational database management system as an abstraction layer over a traditional relational database. Using previously developed use cases that show the feasibility of our framework, we demonstrate the performance of different approaches used in our system.

Keywords variational data, variational RDBMS

ACM Reference Format:
Parisa Ataei, Fariba Khan, and Eric Walkingshaw. 2018. A Variational Database Management System. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

Managing variation in databases is a perennial problem that appears in different forms and contexts [8, 11, 14, 22, 44]. In databases, variation arises when several database instances, which may differ in schema, content, or constraints, conceptually represent the same database. Existing work on database variation focuses on specific kinds of variation such as schema evolution [4, 13, 32, 34, 42], data integration [19], and database versioning [12, 26]. These works provide solutions specific to the kinds of variation they address, but do not provide a general-purpose solution to managing variation in databases. This is a problem since new kinds of variation often arise and different kinds of variation can interact; existing solutions are often ill-suited to these scenarios.

Schema evolution is an example of a kind of variation in databases that is well-supported [4, 13, 32, 34, 42]. In the schema evolution scenario, a database's schema changes over time as the database is extended or refactored to support new information needs, and the goal is to automatically migrate data and queries to new versions of the database. Thus, schema evolution is a kind of *variation* over the dimension of time, and each version of the database can be viewed as a *variant* of the same database.

However, other kinds of database variation are less well supported. One example arises in the context of *software product lines* (SPLs) [3]. An SPL arises when the same code base is used to produce multiple different variants of a software system, customized with different sets of features or tailored for different clients. Naturally, the data requirements of each variant of an SPL may differ [41]. SPL researchers have developed various encodings that allow describing variation in the data model among variants by annotating elements of the model with features from the SPL [1, 40, 41]. These solutions can generate a database schema variant for each software variant of the SPL. However, these solutions address only variation in the data model but do not extend to the level of data or queries. The lack of variation support in queries leads to unsafe techniques such as encoding different variants of query through string munging, while the lack of variation support in data precludes testing with multiple variants of a database at once.

Worse is when multiple kinds of variation interact. Although structural variation over time is well-supported by schema evolution, and structural variation in "space" is partially supported by recent SPL research, there is no support for the inevitable evolution of an SPL's variational data model [24]. Nor do existing approaches support variation across all levels of a relational database: schema, queries, and content. In previous work, we argued that schema evolution, SPL-like variation, and other forms of database variation, such as data integration and database versioning, are all

Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

facets of a similar problem that can be addressed by *treating variation as a general and orthogonal concern* in relational databases [6–8]. An advantage of treating different kinds of variation uniformly is that it is easy to support the interaction of multiple kinds of variation, and to coordinate variation in structure with corresponding variation in queries and content. We illustrate these aspects throughout the paper using a motivating example described in Section 2.

In previous work, we have proposed the idea of *variational databases*, which incorporate variation as a general and orthogonal concern in relational databases [6, 7]. In this paper, we significantly expand on and realize this idea through the formalization and implementation of a *variational database management system* (VDBMS). Specifically, this paper makes the following contributions:

- We provide a formal model of *variational databases* (VDB), whose structure are given by a *variational schema* and whose content are given by *variational tables* (Section 4).
- We define *variational relational algebra* as a query language for VDB, a *static type system* for ensuring that all variants of a query are compatible with the corresponding variants of the VDB, and important properties of VRA (Section 5).
- We implement a prototype of VDBMS as a layer on top of a traditional DBMS (Section 6) and evaluate this implementation on previously developed [8] use cases (Section 7).

2 Motivating Example

In this section, we motivate VDBMS by illustrating the scenario described in Section 1 where two kinds of database variation interact, producing a scenario that is not well supported by any existing tools. The scenario involves the evolution over time of a database-backed SPL.

An SPL uses a set of boolean variables called *features* to indicate functionality that may be included or not in each software variant. Consider an SPL that generates management software for companies. It has a feature *edu* indicating whether a company provides educational resources for its employees. Software variants in which *edu* is enabled (i.e., *edu* = `true`) provide this additional functionality while variants where it is disabled provide only the basic functionality.

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

Each variant of the SPL needs a database to store information about employees, and the selection of features impacts the database: While basic variants do not need to store any education-related records, educational variants do. We visualize the impact of both features and evolution on the schema in Table 1, where feature variation is captured in the columns and variation over time is captured in the rows. Each basic schema variant contains only the schema in a cell in column basic while an educational schema variant consists of two sub-schemas: one from the basic column and another from the educational column. The basic sub-schema and the educational sub-schema may vary over time independently. For example, an extension to the educational sub-schema may use an older version of the basic schema. This is reflected by the highlighted cells in Table 1, which describes a complete schema for a particular educational variant. To describe variation over time of each sub-schema, we introduce two disjoint sets of temporal features (boolean variables, like any other feature), shown in the leftmost and rightmost columns.

Now, consider the following scenario: In the initial design of the basic database, the DBA settles on three tables *empacct*, *job*, *dept*, and *empbio*, shown in Table 1 and associated with feature V_1. After some time, they decide to add attributes *std* and *instr* to the *empacct* table, associated with feature V_1. Since some clients' software relies on a previous design, the two schemas have to coexist in parallel. Therefore, the existence (presence) of *std* and *instr* attributes is *variational*, they only exist in the basic schema when V_2 = `true` and *edu* = `true`. This scenario is an example of *component evolution* in SPLs, where developers update, refactor, and improve components of the SPL, including the database [24].

Now, consider the case where a client that previously requested a basic variant of the software has added courses to educate its employees in specific subjects. The SPL developer needs to enable the *edu* feature for this client, forcing the adjustment of the schema variant to include the educational sub-schema. This case describes *product evolution*, where database evolution in an SPL results from clients adding/removing features/components [24].

The situation is further complicated since the basic and educational schemas are interdependent. Consider the basic schema variant for feature V_2. Attributes *std* and *instr* only exist in the *empacct* relation when *edu* = `true`, represented by a dash-underline, otherwise the *empacct* relation has only the attributes *empno*, *hiredate*, *title*, and *deptno*. Hence, the attributes *std* and *instr* in *empacct* relation are *variational*, that is, they only exist in *empacct* relation when *edu* = `true`.

3 Preliminaries

In this section, we introduce concepts and notations that we use throughout the paper. Throughout the paper, we discuss relational concepts and their variational counterparts. For clarity or when it is unclear from context, we use an *underline* to distinguish a non-variational entity from its variational counterpart, e.g., $\underline{x}$ is a non-variational entity while x is its variational counterpart, if it exists.

3.1 Relational Databases and Relational Algebra

A relational database $\underline{D}$ stores information in a structured manner by forcing data to conform to a *schema* $\underline{S}$ that is a

Table 1. Schema variants of the employee database developed for multiple software variants by an SPL.

Temporal Features	Database Schema Variants for SPL Software Variants		Temporal Features
	basic	educational	
V_1	*empacct* (*empno*, *name*, *hiredate*, *title*, *deptno*) *job* (*title*, *salary*) *dept* (*deptname*, *deptno*, *managerno*) *empbio* (*empno*, *sex*, *birthdate*)	*course* (*courseno*, *coursename*) *teach* (*teacherno*, *courseno*) *student* (*studentno*, *courseno*, *grade*)	T_1
V_2	*empacct* (*empno*, *hiredate*, *title*, *deptno*, *std*, *instr*) *job* (*title*, *salary*) *dept* (*deptname*, *deptno*, *managerno*) *empbio* (*empno*, *sex*, *birthdate*, *name*)	*ecourse* (*courseno*, *coursename*) *course* (*courseno*, *coursename*, *time*, *class*) *teach* (*teacherno*, *courseno*) *student* (*studentno*, *courseno*, *grade*)	T_2
V_3	*empacct* (*empno*, *hiredate*, *title*, *deptno*, *std*, *instr*, *salary*) *dept* (*deptname*, *deptno*, *managerno*, *stdnum*, *instrnum*) *empbio* (*empno*, *sex*, *birthdate*, *firstname*, *lastname*)	*ecourse* (*courseno*, *coursename*, *deptno*) *course* (*courseno*, *coursename*, *time*, *class*, *deptno*) *teach* (*teacherno*, *courseno*) *take* (*studentno*, *courseno*, *grade*)	T_3

finite set $\{\underline{s}_1, \ldots, \underline{s}_n\}$ of *relation schemas.* A relation schema is defined as $\underline{s} = r\left(\underline{a}_1, \ldots, \underline{a}_k\right)$ where each $\underline{a}_i \in \underline{\mathbf{A}}$ is an *attribute* contained in the relation named r. We assume a total order $\leq_{\underline{\mathbf{A}}}$ on $\underline{\mathbf{A}}$, and assume for simplicity that sets of attributes are sorted according to $\leq_{\underline{\mathbf{A}}}$.

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

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

3.2 Encoding Variability

We encode variability in terms of *features.* The *feature space*, $\mathbf{F}$, of a variational database is a closed set of boolean variables called features. A feature $f \in \mathbf{F}$ can be enabled (i.e., $f = \texttt{true}$) or disabled ($f = \texttt{false}$). Features describe the variability in a given variational scenario. For example, in the context of schema evolution, features can be generated from version numbers (e.g., features V_1 to V_3 and T_1 to T_3 in the motivating example, Table 1); for SPLs, the features can be adopted from the SPL feature set (e.g., the *edu* feature in our motivating example, Table 1); and for data integration, the features may represent different data sources.

Features are used at variation points to indicate which variants a particular element belongs to. Thus, enabling or disabling each of the features in the feature set produces a particular database *variant* where all variation has been removed. A *configuration* is a *total* function that maps every feature in the feature set to a boolean value. We represent a configuration by the set of enabled features. For example, in our motivating scenario, the configuration $\{V_3, T_1, edu\}$ represents a database variant where only features V_3, T_1, and *edu* are enabled (and the rest are disabled). This database variant contains relation schemas in the yellow cells of Table 1. We refer to a variant by the configuration that produces it, e.g., variant $\{V_3, T_1, edu\}$ refers to the variant produced by applying that configuration.

When describing variation points in the database, we need to refer to subsets of the configuration space. We do this with propositional formulas of features. Thus, such a propositional formula defines a condition that holds for a subset of configurations and their corresponding variants. For example, the propositional formula $\neg edu$ represents all variants of our motivating example where the *edu* feature is disabled, i.e., variant schemas of the left schema column. We call a propositional formula of features a *feature expression* and define it formally in Figure 1. The evaluation function of feature expressions $\mathbb{E}[\![e]\!]_c : \mathbf{E} \rightarrow \mathbf{C} \rightarrow \mathbf{B}$ simply substitutes each feature f in the expression e with the boolean value given by configuration c and then simplifies the propositional formula to a boolean value. For example, $\mathbb{E}[\![f_1 \vee f_2]\!]_{\{f_1\}} = \texttt{true} \vee \texttt{false} = \texttt{true}$, while $\mathbb{E}[\![f_1 \vee f_2]\!]_{\{\}} = \texttt{false} \vee \texttt{false} = \texttt{false}$. Additionally, in Figure 1, we define a binary *equivalence relation* ($\equiv$) relation on feature expressions corresponding to logical equivalence, and unary *sat*

Feature expression syntax:

$f \in \mathbf{F}$			*Feature Name*
$b \in \mathbf{B}$	$::=$	$\texttt{true} \mid \texttt{false}$	*Boolean Value*
$e \in \mathbf{E}$	$::=$	$b \mid f \mid \neg e \mid e \wedge e \mid e \vee e$	*Feature Exp.*
$c \in \mathbf{C}$	$:$	$\mathbf{F} \rightarrow \mathbf{B}$	*Configuration*

Relations over feature expressions:

$$e_1 \equiv e_2 \textit{ iff } \forall c \in \mathbf{C} : \mathbb{E}[\![e_1]\!]_c = \mathbb{E}[\![e_2]\!]_c$$

$$sat(e) \textit{ iff } \exists c \in \mathbf{C} : \mathbb{E}[\![e]\!]_c = \texttt{true}$$

$$unsat(e) \textit{ iff } \forall c \in \mathbf{C} : \mathbb{E}[\![e]\!]_c = \texttt{false}$$

Figure 1. Feature expression syntax and relations.

and *unsat* predicates that determine whether a feature expression is satisfiable or unsatisfiable, respectively.

To incorporate feature expressions into the database, we *annotate* database elements (including attributes, relations, and tuples) with feature expressions. An *annotated element* x^e consists of an element x and a feature expression e called its *presence condition*. The presence condition determines under which configurations the element is present in the database. For example, assuming $\mathbf{F} = \{f_1, f_2\}$, the annotated number $2^{f_1 \vee f_2}$ is present in variants $\{f_1\}$, $\{f_2\}$, and $\{f_1, f_2\}$ but not in variant $\{\}$. The operation $pc(x^e) = e$ returns the presence condition of an annotated element.

No matter the context, features are often related in ways that constrain the set of possible configurations. For example, only one of the temporal features of V_1–V_3 can be $\texttt{true}$ for a given variant. This relationship is captured by a feature expression, called a *feature model* and denoted by m, which restricts the set of *valid configurations*. That is, a configuration c is only valid if $\mathbb{E}[\![m]\!]_c = \texttt{true}$. For example, the restriction that at a given time only one of temporal features V_1–V_3 can be enabled is represented by the feature model $V_1 \oplus V_2 \oplus V_3$, where $f_1 \oplus f_2 \oplus \ldots \oplus f_n$ is syntactic sugar for $(f_1 \wedge \neg f_2 \wedge \ldots \wedge \neg f_n) \vee (\neg f_1 \wedge f_2 \wedge \ldots \wedge \neg f_n) \vee (\neg f_1 \wedge \neg f_2 \wedge \ldots \wedge f_n)$, that is, the features are mutually exclusive.

3.3 Variational Sets

A *variational set* (*v-set*) $X = \{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}$ is a set of annotated elements [6, 21, 48]. We typically omit the presence condition $\texttt{true}$ in a v-set, e.g., $4^{\texttt{true}} = 4$. Conceptually, a v-set represents many different plain sets simultaneously. Thus, a plain set $\underline{X}$ is a *variant* of the v-set X corresponding to its configuration and it can be generated given the variant $\underline{X}$'s configuration c by evaluating the presence condition of each element with c and including the element if the said evaluation results in $\texttt{true}$ and excluding it otherwise. This is done by the *v-set configuration* function $\mathbb{X}[\![X]\!]_c$. For example, assume we have the feature space $\mathbf{F} = \{f_1, f_2\}$ and the v-set $X_1 = \{2^{f_1}, 3^{f_2}, 4\}$. X_1 represents four plain sets: $\{2, 3, 4\} = \mathbb{X}[\![X_1]\!]_{\{f_1, f_2\}}$, i.e., the variant $\{2, 3, 4\}$ is generated from the v-set X_1 under the configuration $f_1 = \texttt{true}, f_2 = \texttt{true}$; $\{2, 4\} = \mathbb{X}[\![X_1]\!]_{\{f_1\}}$; $\{3, 4\} = \mathbb{X}[\![X_1]\!]_{\{f_2\}}$; and $\{4\} = \mathbb{X}[\![X_1]\!]_{\{\}}$. Following database notational conventions we drop the brackets of a v-set when used in database schema definitions and queries.

A v-set itself can also be annotated with a feature expression. $X^e = \{{x_1}^{e_1}, \ldots, {x_n}^{e_n}\}^e$ is an *annotated v-set* where the feature expression e additionally applies to all elements in the v-set. The *normalization* operation $\downarrow(X^e)$ applies this constraint by pushing it into the presence conditions of the individual elements: $\downarrow(X^e) = \{{x_i}^{e_i \wedge e} \mid x_i^{e_i} \in X^e, sat(e_i \wedge e)\}$. Note that normalization also removes elements with unsatisfiable presence conditions and may be applied to an unannotated v-set X since $X^{\texttt{true}} = X$. For example, in the annotated v-set $X_1 = \{2^{f_1}, 3^{\neg f_2}, 4, 5^{f_3}\}^{f_1 \wedge f_2}$, all elements in the set only exist when both f_1 and f_2 are enabled. Thus, normalizing the v-set X_1 results in $\{2^{f_1 \wedge f_2}, 4^{f_1 \wedge f_2}, 5^{f_1 \wedge f_2 \wedge f_3}\}$. The element 3 is dropped since $\neg sat(pc(3, X_1))$, where $pc(3, X_1) = \neg f_2 \wedge (f_1 \wedge f_2)$. We use the function $pc(x, X^e)$ to return the presence condition of a unique variational element within a bigger variational structure.

We define several operations over v-sets [5]; these operations are used in Section 5.2. The semantics of a v-set operation is equivalent to applying the corresponding plain set operation to every corresponding variant of the argument v-sets. For example, the union of two v-sets $X_1 \cup X_2$ should produce a new v-set X_3 such that $\forall c \in \mathbf{C}.\ \mathbb{X}[\![X_3]\!]_c = \mathbb{X}[\![X_1]\!]_c \mathbin{\underline{\cup}} \mathbb{X}[\![X_2]\!]_c$, where $\underline{\cup}$ is the plain set union operation.

4 Variational Databases

To incorporate variability within a database, we annotate elements with feature expressions, as introduced in Section 3.2. We use annotated elements both in the schema and content. Within a schema we allow attributes and relations to exist conditionally based on the feature expression assigned to them (Section 4.1). At the content level, we annotate each tuple with a feature expression, indicating when the tuple is present (Section 4.2).

4.1 Variational Schemas

A variational schema captures variation in the structure of a database by indicating which attributes and relations are included or excluded in which variants. To this end, we annotate attributes, relations, and the schema itself with feature expressions, which describe the conditions under which each is present. A *variational relation schema* (*v-relation schema*), s, is a relation name with an annotated v-set of attributes, $s \in \mathbf{R} ::= r\,(A)^e$, where A is a v-set of attributes. The presence condition of the v-relation schema, e, determines in what variants of the database the relation itself is present. A *variational schema* (*v-schema*) is an annotated set of v-relation schemas, $S \in \mathbf{S} ::= \{s_1, \ldots, s_n\}^m$. The presence condition of the entire v-schema, m, is the VDB's feature model, which provides a top-level constraint on the set of valid configurations, as described in Section 3.2. Hence, the v-schema defines all valid schema variants of a VDB.

Example 4.1. *S_1 is the v-schema of a VDB that only contains relation empbio in Table 1 and has the feature space* $\mathbf{F} = \{V_1, V_2, V_3\}$. *Note that attributes that exist conditionally are annotated with a feature expression to account for such a condition, e.g., the name attribute only exists when* $V_2 = \texttt{true}$.

$$S_1 = \{empbio(empno, sex, birthdate, name^{V_2}, firstname^{V_3}, lastname^{V_3})\}^{m_1}$$
$$\text{where } m_1 = V_1 \oplus V_2 \oplus V_3.$$

where m_1 allows only one temporal feature for the basic *schema to be enabled at a time.*

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

Similarly, the presence condition of v-relation r is $pc(r, S) = e_r \wedge m$. For example, in Example 4.1 we have $pc(empbio, S_1) = V_3 \wedge m_1$. Furthermore, a database element is only present in a variant for which evaluation of its presence condition under the variant's configuration is `true`, e.g., in Example 4.1 the *name* attribute is present in the variant $\{V_2\}$ because $\mathbb{E}[\![pc(name, S_1)]\!]_{\{V_2\}} = \mathbb{E}[\![V_3 \wedge \texttt{true} \wedge m_1]\!]_{\{V_2\}} = \texttt{true} \wedge \texttt{true} \wedge (\texttt{false} \oplus \texttt{true} \oplus \texttt{false}) = \texttt{true}$ but it is not present in the variant $\{V_3\}$ since $\mathbb{E}[\![pc(name, S_1)]\!]_{\{V_3\}} = \texttt{false} \wedge \texttt{true} \wedge (\texttt{false} \oplus \texttt{false} \oplus \texttt{false}) = \texttt{false}$.

Intuitively and similar to v-sets, a v-schema is a systematic compact representation of a set of plain schemas called variants. A schema variant can be obtained by *configuring* the v-schema with that variant's configuration [5].

4.2 Variational Tables

Variation also exists in database content. To account for content variability, we tag tuples with presence conditions. Thus, a *variational tuple* (*v-tuple*) is an annotated tuple, $u \in U ::= (v_1, \ldots, v_l)^{e_u}$. A v-tuple corresponds to a v-relation, $r(a_1, \ldots, a_l)^{e_r}$, where each element v_i is a value corresponding to attribute a_i (recall that attributes in a v-relation are ordered). For example, $(38, PL, 678)^{T_3}$ is a v-tuple that belongs to the *ecourse* relation from Example 4.1 and is only present when T_3 is enabled. The content of a v-relation is a set of v-tuples, $U \in \mathbf{T} ::= \{u_1, \ldots, u_k\}$ and a *variational table* (*v-table*) is the pair of its relation schema and content, $t = (s, U)$. A *variational database instance* is a set of v-tables, $\mathcal{I} \in \mathbf{I} ::= \{t_1, \ldots, t_n\}^m$. A VDB instance is *well-formed* if its encoded variation at the schema and content level are consistent and satisfiable [8].

Similar to a v-schema, a user can configure a v-table or a VDB for a specific variant [5]. This allows users to deploy a VDB for a specific configuration and generate the corresponding VDB variant. Additionally, our VDB framework puts all variants of a database into one VDB and it keep tracks of which variant a tuple belongs to by annotating them with presence conditions. For example, consider tuples $(38, PL, 678)^{T_3}$ and $(23, DB, \texttt{NULL})^{T_2}$ that belong to the *ecourse* table. The presence conditions T_3 and T_2 state that tuples belong to temporal variants four and five of this VDB, respectively. Hence, this framework tracks which variants a tuple belongs to .

Our VDB framework encodes variation in databases at two levels: schema and content. While content-level variation can stand on its own, such as frameworks used for database versioning and experimental databases [26], the schema level cannot, e.g., $ecourse\left(courseno, coursename, deptno^{T_3}\right)^{edu \wedge (T_2 \vee T_3)}$ encodes variation at the schema level for variants of relation *ecourse* in Table 1. Dropping the presence conditions of tuples leads to ambiguity, i.e., it is unclear which variant each of the tuples $(38, PL, 678)$ and $(23, DB, \texttt{NULL})$ belongs to. We can only guess that they belong to variants where T_2 or T_3 are enabled, but, we do not know for sure which one.

Note that we limit the granularity of content variation to tuples, that is, individual values within a tuple are not variational. This design decision causes some redundancy. For example, the tuples $(38, PL, 678)^{T_3}$ and $(38, PL, \texttt{NULL})^{\neg T_3}$ cannot be represented as a single tuple with variation in the third element. However, this design decision does not prevent us from distinguishing between a `NULL` value that represents a missing value and a `NULL` value that represents a cell that is not present. This distinction can be made by checking the satisfiability of the presence condition of the value v_i in tuple u of relation r in schema S: If $sat(pc(v_i, u))$ then the `NULL` indicates a missing value and otherwise it indicates a non-present cell, where $pc(v_i, u) = e_u \wedge pc(r, S) \wedge pc(att(i), r)$.

5 Variational Queries

The variational nature of a VDB requires a query language that accounts for variation directly. We formally define *variational relational algebra* (VRA) in Section 5.1 as our algebraic query language. A query written in VRA is called a *variational query* (*v-query*), though we often just say *query* when it is clear from context. Unlike relational queries that convey an intent over a single database, a v-query typically conveys the same intent over several relational database variants. However, a single v-query is also capable of capturing different intents over different database variants.

Due to the expressiveness of v-queries, we define a type system for VRA that statically checks a v-query against the underlying v-schema in Section 5.2. To make v-queries more useable we relieve the user from repeating the v-schema's

Operators:

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

Variational conditions:

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

Variational queries:

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

Figure 2. Syntax of variational relational algebra.

variation in their v-queries. This is achieved by explicitly annotating queries in Section 5.3. We then define the *variation-preservation* property for VRA at the type level in Section 5.4.

5.1 Variational Relational Algebra

To account for variation, VRA combines relational algebra (RA) with *choices* [20, 27, 47]. A choice $e\langle x_1, x_2\rangle$ consists of a feature expression e, called the *dimension* of the choice, and two *alternatives* x_1 and x_2. For a given configuration c, the choice $e\langle x_1, x_2\rangle$ can be replaced by x_1 if e evaluates to `true` under configuration c, (i.e., $\mathbb{E}[\![e]\!]_c$), or x_2 otherwise.

Figure 2 defines the syntax of VRA. The selection operation is similar to standard RA selection except that the condition parameter is *variational* meaning that it may contain choices. For example, the query $\sigma_{e\langle a_1=a_2, a_1=a_3\rangle} r$ selects a v-tuple u if it satisfies the condition $a_1 = a_2$ and $sat(e \wedge pc(u))$ or if $a_1 = a_3$ and $sat(\neg e \wedge pc(u))$. The projection operation is parameterized by a v-set of attributes, A. For example, the query $\pi_{a_1, a_2{}^e} r$ projects a_1 from relation r unconditionally, and a_2 when $sat(e)$. The choice operation enables combining two v-queries to be used in different variants based on the dimension. In practice, it is often useful to return information in some variants and nothing in others. We introduce an explicit *empty* query ε to facilitate this. Similar to our definition of the empty query for relational algebra, for VRA we also have: $\varepsilon = \pi_{\{\}} q$. The empty query is used, for example, in q_2 in Example 5.1. The rest of VRA's operations are similar to RA, where all set operations (union, intersection, and product) are changed to the corresponding v-set operations.

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

The result of a v-query is a v-table with the reserved relation name *result*. For example, assume that v-tuples $(1, 2)^{f_1}$ and $(3, 4)^{\neg f_3}$ belong to a v-relation $r(a_1, a_2)$, which is the only relation in a VDB with the trivial feature model `true`. The query $f_3\langle \pi_{a_1 {a_2}^{f_2}} r, \varepsilon\rangle$ returns a v-table with relation schema $result(a_1 {a_2}^{f_2})^{f_3}$, which indicates that the result is only non-empty when f_3 is `true` and that the result includes attribute a_1 when f_2 is `true`. The content of the result relation for the example query is a single v-tuple $(1)^{f_1}$. The tuple $(3)^{\neg f_3}$ is not included since the projection occurs in the context of a choice in f_3, which is incompatible with the presence condition of the tuple, i.e., $unsat(f_3 \wedge \neg f_3)$. This illustrates how choices can effectively filter the tuples in a VDB based on the dimension. Example 5.1 illustrates how a v-query can be used to express variational information needs.

Example 5.1. *Assume we have the VDB introduced in Example 4.1. The user wants the employee ID numbers (empno) and names for variants $\{V_2\}$ and $\{V_3\}$. The user needs to project the name attribute for variant $\{V_2\}$, the firstname and lastname attributes for variant $\{V_3\}$, and empno attribute for both variants. This can be expressed with the following v-query.*

$$q_1 = \pi_{empno^{V_2 \vee V_3}, name, firstname, lastname}\, empbio$$

In the example, note that the user does not need to repeat the variability encoded in the v-schema in their query, that is, they do not need to annotate *name*, *firstname*, and *lastname* with V_2, V_3, and V_3, respectively. We discuss this in more detail in Section 5.3. q_1 queries all three variants simultaneously although the returned results are only associated with variants V_2 and V_3 due to the annotation of the attribute *empno* in the query and the presence conditions of the rest of the projected attributes in the schema. Yet, the query can be more simplified with a choice. q_2 selects only two out of the three variants explicitly: $q_2 = V_2 \vee V_3\langle \pi_{empno, name, firstname, lastname}\, empbio, \varepsilon\rangle$. Note that queries q_1 and q_2 return the same set of v-tuples since neither returns tuples associated with variant V_1, but their returned v-tables have different presence conditions, thus, q_2 filters out tuples that belong to variant V_1 while q_1 does not.

Expressing the same intent over several database variants by a single query relieves the DBA from maintaining separate queries for different variants or configurations of the schema. Example 5.2 illustrates this point.

Example 5.2. *Assume a VDB with* $\mathbf{F} = \{V_1, V_2, V_3\}$ *and the corresponding* basic *schema variants in Table 1. The user wants to get all employee names across all variants. They express this intent by the query q_3:*

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

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

Figure 3. Configuration of VRA. It assumes that the v-query is well-typed. $\mathbb{R}[\![.]\!]_c$, $\mathbb{C}[\![.]\!]_c$, and $\mathbb{A}[\![.]\!]_c$ are configuration of v-relation, v-condition, and v-set of attributes, respectively. This function is a trivial recursion for the rest of VRA operations and $\mathbb{Q}[\![\varepsilon]\!]_c = \varepsilon$.

Since the v-schema enforces that exactly one of $V_1 - V_3$ be enabled, we can simplify the query by omitting the final choice.

$$q_4 = V_1\langle \pi_{name} empacct, \pi_{name, firstname, lastname} empbio\rangle$$

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

The *configuration* function maps a v-query under a configuration to a relational query, defined in Figure 3. Thus, a v-query can be understood as a set of relational queries that their results is gathered in a single table and tagged with the feature expression stating their variant. Users can deploy queries for a specific variant by configuring them. Example 5.3 illustrates configuring a query.

Example 5.3. *Assume the underlying VDB has the v-schema $S_3 = \{r\,(a_1{}^{f_1}, a_2, a_3)^{f_1 \vee f_2}\}$ and $\mathbf{F} = \{f_1, f_2\}$. The v-query $q_5 = \pi_{a_1, a_2{}^{f_1 \wedge f_2}, a_3{}^{f_2}} r$ is configured to the following relational queries: $\mathbb{Q}[\![q_5]\!]_{\{f_1\}} = \mathbb{Q}[\![q_5]\!]_{\{\,\}} = \pi_{\underline{a_1}}\underline{r}$, $\mathbb{Q}[\![q_5]\!]_{\{f_2\}} = \pi_{\underline{a_1}, \underline{a_3}}\underline{r}$, $\mathbb{Q}[\![q_5]\!]_{\{f_1, f_2\}} = \pi_{\underline{a_1}, \underline{a_2}, \underline{a_3}}\underline{r}$.*

VRA enables querying multiple database variants encoded as a singled VDB simultaneously and selectively. More precisely, VRA is *maximally expressive* in the sense that it can express any set of plain RA queries over any subset of relational database variants encoded as a VDB. We prove this claim in Theorem 5.4.

Theorem 5.4. *Given a set of plain RA queries $\underline{q}_1, \ldots, \underline{q}_n$ where each query $\underline{q}_i$ is to be executed over a disjoint subset $\mathcal{I}_i$ of variants of the VDB instance $\mathcal{I}$, there exists a v-query q such that $\forall c \in \mathbf{C}.\ \mathbb{I}[\![\mathcal{I}]\!]_c = \mathcal{I}_i \implies \mathbb{Q}[\![q]\!]_c = \underline{q}_i$.*

Proof. By construction. Let f_i be the feature expression that uniquely characterizes the variants in each $\mathcal{I}_i$. Then $q = (f_1 \wedge \neg f_2 \wedge \ldots \wedge \neg f_n)\langle \underline{q}_1, (f_2 \wedge \ldots \wedge \neg f_n)\langle \underline{q}_2, \ldots f_n\langle \underline{q}_n, \varepsilon\rangle \ldots\rangle\rangle$. □

The above construction relies on the fact that every RA query is a valid VRA (sub)query in which every presence condition is `true`. Of course, in most realistic scenarios, we expect that v-queries can be encoded more efficiently by sharing commonalities and embedding relevant choices and presence conditions within the v-query.

5.2 Well-Typed Queries

In this section, we introduce a static type system for VRA. The type system ensures that queries are consistent with the underlying variational schema. That is, that all referenced relations and attributes are present in the variation contexts in which they are used. For example, consider the VDB from Example 5.3 that contains only the relation $r(a_1{}^{f_1}, a_2, a_3)^{f_1 \vee f_2}$. The query $\pi_{a_4} r$ is ill-typed since a_4 is not present in r. Similarly, the queries $\pi_{a_1{}^{\neg f_1}} r$ and $f_1\langle \pi_{a_2} r, \pi_{a_1} r\rangle$ are both ill-typed since a_1 is not present in r when f_1 is disabled.

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

Figure 4 defines a typing relation that relates VRA queries to their types. The judgment form $e, S \vdash q : A^{e'}$ states that in variation context e within v-schema S, v-query q has type $A^{e'}$. If a query does not have a type, it is *ill-typed*. A *variation context* is a feature expression that tracks which variants the current subquery is present in. We sometimes use the judgment form $S \vdash q : A^{e'}$ when the variation context is the unextended feature model, that is, $pc(S), S \vdash q : A^{e'}$. We assume that the v-set of attributes A is normalized to remove elements with unsatisfiable presence conditions, but this normalization is only shown explicitly in the rules where strictly necessary.

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

For a projection $\pi_A q$, the rule Project-E checks that all projected attributes A are present in at least one variant of the

VRA typing rules:

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

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

$$\textsc{Select-E}\quad \frac{e, S \vdash q : A^{e'} \qquad e, \downarrow(A^{e'}) \vdash \theta}{e, S \vdash \sigma_\theta q : A^{e'}}$$

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

$$\textsc{EmptyRelation-E}\quad e, S \vdash \varepsilon : \{\,\}^{\texttt{false}}$$

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

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

V-condition typing rules: (As a reminder, b: boolean tag, $\underline{a}$: plain attribute, k: constant value):

$$\textsc{Boolean-C}\quad e, A \vdash b$$

$$\textsc{AttOptVal-C}\quad \frac{a^{e'} \in A \qquad sat(e' \wedge e)}{e, A \vdash \underline{a} \bullet k}$$

$$\textsc{AttOptAtt-C}\quad \frac{{a_1}^{e_1} \in A \qquad {a_2}^{e_2} \in A \qquad sat(e_1 \wedge e_2 \wedge e)}{e, A \vdash \underline{a}_1 \bullet \underline{a}_2}$$

$$\textsc{Neg-C}\quad \frac{e, A \vdash \theta}{e, A \vdash \neg\theta}$$

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

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

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

Figure 4. The rules assume that the underlying VDB is well-formed. Remember that our theory assumes all attributes have the same type and all constants belong to attributes' domain.

variation context (second premise) and that these attributes are *subsumed* by type of the subquery q (third premise). The subsumption relation $A \prec A'$ used in the third premise is defined as $\forall a^{e_1} \in \downarrow(A). \exists e_2. a^{e_2} \in \downarrow(A'), sat(e_1 \wedge e_2)$, which ensures that all of the projected attributes are present in the type of the subquery q, and that the presence conditions of the v-set of projected attributes do not contradict the presence conditions in the type of q. The result type is the v-set intersection of the projected attributes and the attributes of the subquery ensuring that the variation constraints of both are captured.

The rule Select-E checks if its subquery and v-condition are well-typed and if so it returns the subquery's type. The v-condition typing relation is defined in Figure 4 and has the judgment form $e, A \vdash \theta$, which states that the v-condition θ is well-formed in variation context e within attribute v-set A. The v-condition typing rules ensure that each attribute used in a v-condition is present in A and that the presence condition associate with that attribute does not contradict the current variation context.

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

The EmptyRelation-E rule states that an empty relation has the type of an empty set annotated by `false`, which is the required base case to ensure that the type system is variation preserving (see Section 5.4). The remaining rules are straightforward extensions of the standard relational algebra typing rules for product and set operations to account for variation contexts and v-sets.

5.3 Explicitly Annotating Queries

V-queries do not need to repeat information that can be inferred from the v-schema or the type of a query. For example, the query q_1 shown in Example 5.1 does not contradict the schema and thus is type correct. However, it does not include the presence conditions of attributes and the relation encoded in the schema while q_6 repeats this information:

$$q_6 = \pi_{empno^{V_2 \vee V_3}, name^{V_2}, firstname^{V_3}, lastname^{V_3}} \left(m_2\langle empbio, \varepsilon\rangle\right) .$$

Similarly, the projection in the query $q_7 = \pi_{name, firstname} subq_7$ where $subq_7 = V_2\langle \pi_{name} q_6, \pi_{firstname} q_6\rangle$ is written over S_1 and it does not repeat the presence conditions of attributes from its $subq_7$'s type. The query $q_8 = \pi_{name^{V_2}, firstname^{\neg V_2}} subq_7$ makes the annotations of projected attributes *explicit* w.r.t. both the v-schema S_1 and its subquery's type. Although relieving the user from explicitly repeating variation makes

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

Figure 5. Explicitly annotating a well-typed query with a v-schema. This function is a trivial recursion for the rest of VRA operations and $\lfloor \varepsilon \rfloor_S = \varepsilon$.

VRA easier to use, queries still have to state variation explicitly to avoid losing information when decoupled from the schema. We do this by defining the function $\lfloor q \rfloor_S : \mathbf{Q} \rightarrow \mathbf{S} \rightarrow \mathbf{Q}$, that *explicitly annotates a query q with the schema S*. The explicitly annotating query function, formally defined in Figure 5, conjoins attributes and relations presence conditions with the corresponding annotations in the query and wraps subqueries in a choice when needed. Note that, q_8 and q_6 are $\lfloor q_7 \rfloor_{S_1}$ and $\lfloor q_1 \rfloor_{S_1}$, respectively, after simplification.

Theorem 5.5. *If the query q has the type A then its explicitly annotated counterpart has the same type A, i.e.:*

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

This shows that the type system applies the schema to the type of a query although it does not apply it to the query. The type equivalence *is v-set equivalence for normalized v-set of attributes.*

We have proved Theorem 5.5 in the Coq proof assistant [31].

5.4 Variation-Preservation Property

To show that VRA is type safe we benefit from RA's type safety [36] by defining the *variation-preservation* property for VRA which connects VRA to RA. The variation-preservation property states that if a query q has type A then configuring the type of a valid explicitly annotated query is the same as the type of its configured corresponding query. Theorem 5.6 proves this property.

$$\begin{tikzcd}
\lfloor q \rfloor_S \arrow[r, "type"] \arrow[d, dashed, "\mathbb{Q}[\![.]\!]_c"'] & A^e \arrow[d, dashed, "\mathbb{A}[\![.]\!]_c"] \\
\underline{q} \arrow[r, "\underline{type}"] & \underline{A}
\end{tikzcd}$$

Theorem 5.6 is visualized by the commuting diagram below, where the vertical arrows indicate corresponding configure functions, *type* indicates VRA's type system, i.e., $type(q) = A^e$ is $S \vdash q : A^e$, and $\underline{type}$ indicates RA's type system, i.e., $\underline{S} \vdash \underline{q} : \underline{A}$. We assume that the v-schema and schema are passed to the respective type systems. The diagram states that if we configure q with c, then determine the type of the plain query $\underline{q}$ using the standard RA type system (the lower left path), we get the same type $\underline{A}$ as if we had instead inferred a variational type A for q using the VRA type system, then configure A with c (the upper right path). The variation-preservation property ensures that the type of a variational query encodes the types of all the plain queries it encodes.[1] The query must be explicitly annotated since the configuration function for v-queries does not take the v-schema into account.

Theorem 5.6. *For all configurations c, if a query q has type A then its configured query $\mathbb{Q}[\![\lfloor q \rfloor_S]\!]_c$ has type $\mathbb{A}[\![A]\!]_c$, i.e.,*

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

Proof. By structural induction. We proved this theorem in the Coq proof assistant [31]. □

6 VDBMS Implementation

We implement a prototype of the VDB and VRA frameworks as the *Variational Database Management System (VDBMS)*. VDBMS is implemented in Haskell and sits on top of any standard relational DBMS. Data is stored in the form of v-tables, explained in Section 4.2. The presence conditions on tuples are stored as an additional attribute in each table. All of the other presence conditions in our framework are stored either in the Haskell layer or in a separate metadata table in the underlying database. To support running VDBMS with multiple different plain relational DBMS backends, we provide a shared interface for communicating with the backend DBMS and instantiate it for different database engines such as PostgreSQL and MySQL. An expert can extend VDBMS to another database engine by writing methods for connecting to and querying the database.

6.1 VDBMS Architecture

Figure 6 shows the architecture of VDBMS. The VDB and its v-schema are stored in the DBMS, shown in the bottom right. A VDB can be *configured* to its plain relational database variants for deployment. For example, an SPL developer may configure a VDB to produce the plain relational database to accompany a software product for a client.

To extract information from a VDB, a user inputs a v-query q to VDBMS. First, q is checked by the *type system*. If the query is ill-typed, the user gets an error explaining what part of the query violated the v-schema. Otherwise, q is explicitly annotated with information from the schema and passed to the *variation minimization* module. This module simplifies the query based on syntactic rules [5] that preserve the semantics of the query. These rules make VRA flexible since an information need can be represented by multiple different v-queries as demonstrated in Example 5.1 and Example 5.2. The simplified query is then sent to the *generator* module where SQL queries are generated from v-queries by different approaches explained in Section 6.2. The generated SQL queries are then executed on the underlying DBMS yielding one or more v-tables. These are passed to the *v-table builder*, which combines them into a single v-table to be returned to

[1]We have also defined this property at the semantics level to show that *running* a variational query corresponds to running all of its variants on all of the variants of the VDB [5].

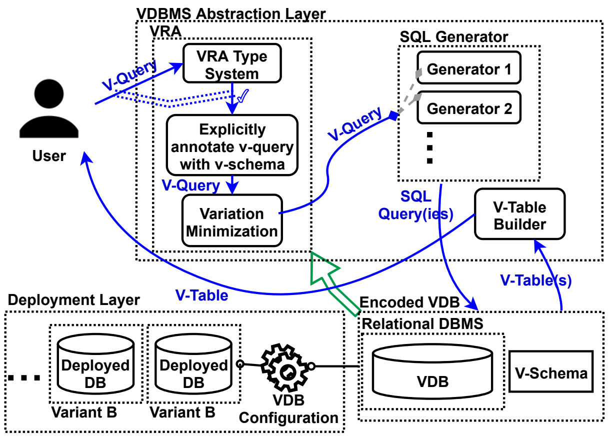


Figure 6. VDBMS architecture. The dotted double-line from the input v-query indicates the dependency of passing the v-query to this module only if it is well-typed.

the user. The v-table builder cleans up the resulting v-table by filtering out duplicate and invalid tuples, and shrinking presence conditions.

6.2 SQL Generators

Since VDBMS sits on top of a plain relational DBMS, v-queries must be translated into (sets of) plain relational queries. We have explored multiple different *SQL generators* for v-queries. In the descriptions below, we focus on the differences among each approach. However, each of these generators must also add relation qualifiers to attributes, rename subqueries where appropriate, and extend the list of projected attributes to include the special presence condition attribute stored with each v-table. The resulting v-tables produced by each approach are passed to the v-table builder to be combined into a single resulting v-table.

Given an explicitly annotated, well-typed v-query q, we explored five approaches to generate SQL queries for q:

1. *Naive Brute Force (NBF)*: Configures q into a set of RA queries corresponding to each valid configuration of q.
2. *Unique Brute Force (UBF)*: Like NBF except that we only generate SQL queries for unique RA queries produced by configuring q.
3. *Union-All-Variants (UAV)*: Takes the SQL queries generated by UBF and unions them into a single SQL query. To do so, it forces all of the SQL queries to return the same relation schema. Additionally, it applies the presence condition of each SQL query to its tuples by concatenating it with the presence condition attribute in the projected attribute set.
4. *Injected Naive Brute Force (NBF(i))*: Similar to NBF, except that the feature expression associated with each configuration is injected into the SQL query. This simplifies the job of the v-table builder, which must only fix the schema of the returned tables rather than modifying the presence conditions of all returned tuples.
5. *Injected Unique Brute Force (UBF(i))*: The same idea as NBF(i), except applied to the UBF approach.

The SQL generators produce queries in a generic query format that can be rendered into concrete SQL queries executable on each underlying DBMS engine. Each of these approaches conduct the same essential tasks. However, they vary in where they enforce variation preservation. For example, UAV, NBF(i), and UBF(i) generate queries that preserve variation by construction, while NBF and UBF yield intermediate v-tables that violate variation preservation and then recover this property in the v-table builder module. A complete description of each of the approaches, and examples of the SQL queries generated for each, can be found in [5].

7 Experiments and Discussion

In this section, we compare the performance of VDBMS with the different SQL generation approaches described in Section 6.2. Note that this is not intended as a comprehensive empirical evaluation. The main contributions of this work are the new functionality that VDBMS adds to traditional RDBMSs, the formal model of variational databases described in Section 4, and the theoretical results described in Section 5. These experiments are rather provided as a demonstration that our approach works as described and to facilitate discussion of implementation design decisions.

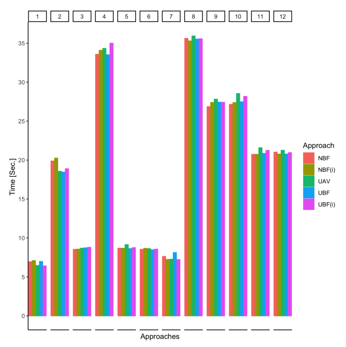
For our experiments, we use two previously developed datasets [8], both of which are available[2] online:

1. An *email SPL* VDB, which illustrates database variation for an email system SPL. The feature model is taken from [23] with some simplifications, resulting in eight features with five database variants. The data is taken from the Enron corpus[3] and is adapted based on the needs of the SPL. The v-queries were written to resolve the feature interactions in the email SPL described in [23].
2. An *employee evolution* VDB, which illustrates the application of VDBMS to schema evolution. The schema, its evolution, and queries are taken from [34] with some adaptation. It has five features denoting each time the schema evolved, which results in five database variants. The data is from a real-world employee database.[4]

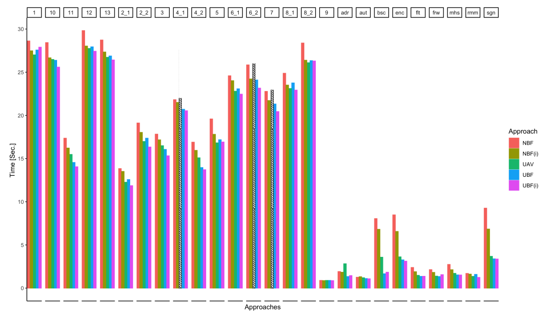
We use these datasets to ensure the correctness of each approach and compare them. To ensure our SQL generator methods are implemented correctly we conducted two *sanity checks* for all v-queries that yield runnable SQL queries, which passed for every combination of approach and dataset:

1. We check that the variation-preservation property holds. That is, we check that for all configurations, configuring the

[2]https://zenodo.org/record/4321921
[3]http://www.ahschulz.de/enron-email-data/
[4]https://github.com/datacharmer/test_db



(a) The employee evolution VDB.



(b) The email VDB.

Figure 7. Comparison of SQL generators NBF, NBF(i), UAV, UBF, and UBF(i).

resulting v-table is the same as running an identically configured query over the corresponding configured database.

2. We check that the results from each pair of methods are equivalent.

We compare the runtime of each of our approaches to executing queries. The runtime contains all elements of executing a query, including type checking, annotation, optimization, SQL generation, and v-table building. We run the experiments on a MacBook Pro with a 2.4 GHz Core i7 processor and 8 GB of 1600 MHz DDR3 RAM. All experiments are run with PostgreSQL 13.3 as the database engine.

Figure 7 shows the runtime for each query for each of the introduced approaches in Section 6.2 over the employee and email VDBs. The queries are labeled at the top of the plots and the approaches are indicated by different color bars.

Figure 7a shows that NBF(i) is usually faster than NBF, but UBFand UBF(i) do not follow a similar pattern. Additionally, NBF(i), UAV, and UBF are close in performance and none consistently performs better than the others.

Figure 7b also shows that NBF(i) is consistently faster than NBF, and UBF(i) is also mostly faster than UBF for this dataset. While UAV mostly performs better than NBF(i), it is comparable to UBF(i) for this dataset. Yet, UAV sometimes generates a non-runnable SQL query due to forcing an attribute to have type string when it is projected as `NULL` [5]. This issue can be addressed by forcing the first subquery of the union to have all attributes projected and limiting the number of returned tuples to zero. This would force the type of attributes as they are in the schema and since it does not return any tuples it will not change the result.

Based on our experiments, the query construction (from type system to generating SQL queries) takes similar time between the approaches. Their main difference comes down to the gross runtime of queries on the VDB and building the v-table. UAV does not take any time to build a v-table since the result already has the desired schema and presence conditions, however, it spends more time on running the SQL query since queries generated by UAV are usually more complicated. On the other hand, although NBF and UBF run multiple SQL queries per v-query their generated SQL queries are simpler than the ones generated by UAV. However, in contrast to UAV, they must adjust the returned table for each SQL query and apply the correct presence condition to the tuples. Finally, the main difference between the performance of NBF and NBF(i) (and similarly, UBF and UBF(i)) is where they apply the correct presence condition to the tuples. While NBF(i) and UBF(i) pass this task to the underlying database engine (which seems to perform better) the NBF and UBF approaches do this task in the v-table builder, in the Haskell layer. Note that all four of these approaches still have to fix the schema of the returned tables to the v-table schema of the v-query.

8 Related Work

While variational databases provide a *generic* solution to the problem of variation in databases, lots of databases research has addressed *specific kinds* of variation. As described in Section 1, these more focused approaches often provide better solutions to one kind of variation but are less flexible and so not suitable for many applications. In the rest of this section, we briefly describe this databases research, and also describe previous work from the SPL community.

Schema evolution: Current solutions addressing schema evolution rely on temporal nature of schema evolution. They use timestamps as a means to track historical changes either in an external document [34] or as versions attached to databases [4, 13, 32, 42]. Some require the DBA to design a unified schema to support querying different versions [25], while others support migrating database variants to alternate versions [4, 13, 32, 42]. A third approach is to transform queries written against one version to other versions [34]. Temporal evolution is tracked by requiring the database to always have a time-related attribute in tables. Thus, queries have to specify the time frame for which they are inquiring information [34]. Now the user can choose a wide enough time frame in their queries to access to their desired variant(s). Unlike variational databases, schema evolution does not support expressing a variation need against two different variants at once, nor does it support expressing multiple information needs in a single query.

Data integration: Data integration systems need to query disparate data sources which often have different formats and have been developed independently of each other [19]. Thus, work on *data integration* can be viewed as managing database variation in space at the content, schema, and format level. The ability to combine data in different formats (e.g., relational and unstructured data) is not supported by variational databases. Most data integration systems fall somewhere on the spectrum of warehousing and virtual integration. In the warehousing model, data from each source is loaded and materialized in a physical database called a warehouse whereas in the virtual integration model, the data remain in each data source and are accessed as needed at query time. The VDB framework falls in the middle of the spectrum since all database variants (data sources) are stored in a physical database without materialization.

In data integration systems, a *mediated schema* is defined for the integrated data and each data source has a wrapper/extractor that adjusts the schema and data of the source to the mediated schema [10, 18, 38]. Users query against the mediated schema, and the system assembles the results from the various sources. The set of wrapper/extractors collectively describe the variation among integrated systems, filling a role similar to our v-schemas.

A concern in data integration is tracking *data provenance*, that is, which sources the results of a query came from. Data provenance can be realized in variational databases by attaching a different feature to the tuples in each source. The variation preservation property ensures these features will be propagated to the query results.

Temporal databases and database versioning: Temporal databases [29, 37, 45] address scenarios where the state of the data at a specific time is important, such as financial and medical data. Some temporal databases extend traditional relational databases [2, 39, 46], while others adopt an in-database approach [30]. Temporal databases contain timestamps as one kind of value and queries may express conditions over timestamps [17, 28, 33]. Temporal databases only support content-level variation and are therefore less expressive than variational databases in terms of the variation that they can describe. However, their special support for temporal values and conditions make expressing temporal constraints easier than an equivalent encoding in VDB/VRA, where one would have to introduce features for each timestamp and then use potentially large feature expressions to describe temporal ranges.

Database versioning aims to support collaborative data curation and analysis [11] by supporting non-linear, temporal changes to a database. Once such example is *OrpheusDB* [26], which is inspired by version control systems like Git. It captures different evolutions of the database in a version graph. A version graph can be viewed as a kind of variational database, although only one particular variant can be checked out at a time. OrpheusDB supports both git-style version control commands and SQL-like queries. Its query language *VQL* can query the data as well as their versions. VQL supports a subset of the query language for versioning and provenance proposed by Chavan et al. [15]. Bhattacherjee et al. [12] studies the trade-off of storage and recreation cost for different compression and optimization methods used to recreate a database version. Similar to temporal databases, database versioning systems only support content-level variation.

Database variability in software development: A database may contain variation due to evolution of the business requirements of its associated software [24, 43]. Schema evolution and database migration can sometimes be used to address this kind of database variation, but other workarounds have been proposed as well. The first is that a different relational database may be *specified and created per-variant*, according to the information needs of each variant [35]. This is labor-intensive and difficult to maintain since changes need to be propagated across variants manually. The second strategy is to define a single *global schema that applies to all variants* [9]. This strategy is more efficient to maintain, but can be complex and error prone in som some scenarios, such as SPL evolution [40, 41]. The third strategy is to define a *variable data model* [1, 40, 41] which models a database schema with annotations of features from the SPL to indicate their variable existence.

9 Conclusion and Future Work

We presented the variational database framework as a generic solution to encode and query multiple variants of a database as a single entity. The VDB framework systematically ensures that the variation in data and queries are encoded correctly and consistently, removing the burden of manual workarounds from database administrators and developers.

The VDB framework assumes a closed-world variation and configuration space, that is, the sets of features and configurations are closed. An immediate improvement would be to make the configuration and feature-space dynamically extensible. This is an essential piece to updating VDBs, which we do not yet address, since it would enable incrementally adding new variants to a VDB. Additionally, the performance of VDBMS can be improved by different optimizations, such as defining a user-defined type and functions for presence conditions in databases.

More speculatively, VDBMS could be made more usable by providing a visual interface that shows a snapshot of the database as the user writes their query. This improvement requires a type system that allows for holes in queries. This can be enriched by designing an error-tolerant type system that pinpoints where the user made a mistake in their query and allows the part of query that is well-typed to run [16].

Finally, although we have not implemented a system to generate a VDB for a variation scenario, it is trivial to do so if we have the variant databases. The problem is that, in most cases, the variant databases do not exist since current variation scenarios only simulate the effect of variation and do not incorporate it directly into the database or queries. Thus, an expert needs to manually generate the database variants. Another way to do this would be to extend the query language to support database updates.

References

[1] Lamia Abo Zaid and Olga De Troyer. Towards modeling data variability in software product lines. In Terry Halpin, Selmin Nurcan, John Krogstie, Pnina Soffer, Erik Proper, Rainer Schmidt, and Ilia Bider, editors, *Enterprise, Business-Process and Information Systems Modeling*, pages 453–467, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-21759-3.

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

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

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

[5] Parisa Ataei. *The Theory and Implementation of a Variational Database Management System.* PhD thesis, Oregon State University, 2021. Draft.

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

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

[8] Parisa Ataei, Qiaoran Li, and Eric Walkingshaw. Should variation be encoded explicitly in databases? In *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS'21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450388245. doi: 10.1145/3442391.3442395. URL https://doi.org/10.1145/3442391.3442395.

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

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

[11] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings.* www.cidrdb.org, 2015. URL http://cidrdb.org/cidr2015/Papers/CIDR15_Paper18.pdf.

[12] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow.*, 8(12):1346–1357, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824035. URL http://dx.doi.org/10.14778/2824032.2824035.

[13] Cristina De Castro, Fabio Grandi, and Maria Rita Scalas. Schema versioning for multitemporal relational databases. *Information Systems*, 22(5):249 – 290, 1997. ISSN 0306-4379. doi: https://doi.org/10.1016/S0306-4379(97)00017-3. URL http://www.sciencedirect.com/science/article/pii/S0306437997000173.

[14] Badrish Chandramouli, Johannes Gehrke, Jonathan Goldstein, Donald Kossmann, Justin J. Levandoski, Renato Marroquin, and Wenlei Xie. READY: completeness is in the eye of the beholder. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* www.cidrdb.org, 2017. URL http://cidrdb.org/cidr2017/papers/p18-chandramouli-cidr17.pdf.

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

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

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

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

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

[20] 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.

[21] Martin Erwig, Eric Walkingshaw, and Sheng Chen. An Abstract Representation of Variational Graphs. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2013.

[22] Mina Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. Clams: Bringing quality to data lakes. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2089–2092, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2899391. URL https://doi.org/10.1145/2882903.2899391.

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

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

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

[26] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya Parameswaran. Orpheusdb: Bolt-on versioning for relational databases. *Proc. VLDB Endow.*, 10(10):1130–1141, June 2017. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=3115404.3115417.

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

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

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

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

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

[32] Edwin McKenzie and Richard Thomas Snodgrass. Schema evolution and the relational algebra. *Inf. Syst.*, 15(2):207–232, May 1990. ISSN 0306-4379. doi: 10.1016/0306-4379(90)90036-O. URL http://dx.doi.org/10.1016/0306-4379(90)90036-O.

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

[34] Hyun J. Moon, Carlo A. Curino, Alin Deutsch, Chien-Yi Hou, and Carlo Zaniolo. Managing and querying transaction-time databases under schema evolution. *Proc. VLDB Endow.*, 1(1):882–895, August 2008. ISSN 2150-8097. doi: 10.14778/1453856.1453952. URL http://dx.doi.org/10.14778/1453856.1453952.

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

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

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

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

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

[40] Martin Schäler, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Building information system variants with tailored database schemas using features. In Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanislaw Wrycza, editors, *Advanced Information Systems Engineering*, pages 597–612, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-31095-9.

[41] Norbert Siegmund, Christian Kästner, Marko Rosenmüller, Florian Heidenreich, Sven Apel, and Gunter Saake. Bridging the Gap Between Variability in Client Application and Database Schema. In *13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 297–306. Gesellschaft für Informatik (GI), 2009.

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

[43] Micheal Stonebraker, Dong Deng, and Micheal L. Brodie. Database decay and how to avoid it. In *Big Data (Big Data), 2016 IEEE International Conference*. IEEE, 2016. doi: 10.1109/BigData.2016.7840584.

[44] U. Störl, D. Müller, A. Tekleab, S. Tolale, J. Stenzel, M. Klettke, and S. Scherzinger. Curating variational data in application development. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1605–1608, 2018. doi: 10.1109/ICDE.2018.00187.

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

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

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

[48] 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.