# Variational Databases

Parisa Ataei
Oregon State University
ataeip@oregonstate.edu

Arash Termehchy
Oregon State University
termehca@oregonstate.edu

Eric Walkingshaw
Oregon State University
walkiner@oregonstate.edu

## ABSTRACT

Data variations are prevalent in real-world applications. For example, software vendors handle variations in the business requirements, conventions, and environmental settings of a software product using hundreds of features each combination of which creates a different version of the product. In database-backed software, the database of each version may have a different schema and different content. Variations in the value and representation of each element in a dataset give rise to numerous variants in these applications. Users often would like to express information needs over all such variants. For example, a software vendor would like to perform common tests over all versions of its product, e.g., whether each relation in a relational database has a primary key. Hence, users need a query interface that hides the variational nature of the data and processes a query over all variations of a dataset. We propose a novel abstraction called a *variational database* that provides a compact and structured representation of general forms of data variations and enables users to query database variations easily.

## KEYWORDS

Data Variation, Variational Schema, Variational Database

## 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 needs 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 [1]. 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 databases 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. As 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.

In these applications, users would like to work with a query interface 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 over such abstraction must be *variation-preserving*: their inputs and outputs are sets of database variants. Example 1.1 demonstrates the need for a system that can query multiple databases given a query on only one of them.

*Example 1.1. Anthropometric data* [1] are measurements of human bodies which help industrial designers build precise and usable products. Anthropometric data is collected by physicians around the globe working at different institutes, often as part of different experiments and to address different questions. Since human physiology differs significantly, designers want data from diverse and representative populations. Each designer, however, needs a different sets of attributes based on her application domain. Consider a software production company that produces softwares to manage these datasets. The company may produce different versions of its software for each application domain and environmental setting. For instance, the version of the software and its database used for the automotive application domain is different with the one used in designing computers. Table 1 shows two possible schemas used in two different versions of the software.

Views have traditionally been used to define unified abstractions over multiple databases. Nevertheless, queries over views are not variation-preserving and their output is a single relation. 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 paper, we outline our proposal for a novel abstraction called *variational databases* that provides a unified, compact, and structured representation of data variants in an application domain. We also define the concept of *variational queries* whose inputs and outputs are variational databases. One may use variational queries

[1] http://mreed.umtri.umich.edu/mreed/downloads.html


*DBPL 2017, September 1, 2017, Munich, Germany*

ACM ISBN 978-1-4503-5354-0/17/09…$15.00
https://doi.org/10.1145/3122831.3122839

| Schema Variant 1 | Schema Variant 2 |
|---|---|
| info(id, gender, date, physician) | subjectInfo(id, gender) |
| hand(hand_lnth, hand_brth) | physician(id, name, date) |
| foot(foot_circ, foot_lnth) | measurement(hand_lnth, foot_lnth) |

**Table 1: Schemas of two datasets for anthropometric data.**

$$
\begin{array}{rcl}
o \in \mathit{Opt} & & \\
b \in \mathit{Bool} & ::= & \texttt{true} \mid \texttt{false} \\
c \in \mathit{Cond} & ::= & b \mid o \mid \neg c \mid c_1 \vee c_2 \mid c_1 \wedge c_2
\end{array}
$$

**Figure 1: Presence conditions.**

to explore database variants simultaneously while preserving differences between them. In this paper, we mainly focus on schematic variations in data, however, we believe that variational databases have the potential to handle other types of data variations.

## 2 PRELIMINARIES

A schema $S$ is a finite set $\{R_1, \ldots, R_n\}$ of relation symbols where each $R_i$ has a fixed arity $n_i \geq 0$. 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 simplicity, we assume that all attributes in a schema share the same domain. Our definitions can be naturally extended for the case where attributes may not share the same domain. The domain $dom(I)$ of instance $I$ is the set of all constants that occur in at least one relation in $I$. We denote all instances of schema $S$ as $Inst(S)$. A $m$-ary query $q$ over a schema $S$, $m \geq 0$, is a function that maps every instance $I \in Inst(S)$ into a relation with arity $m$ $q(I) \subseteq dom(I)^m$.

## 3 REPRESENTATION

In this section we define variational schemas and variational queries. The definitions are based on generic representations of variational sets and maps, described in Section 3.1. Variational schemas and queries are defined in Section 3.2 and Section 3.3, respectively. The process of lifting a plain query written against a plain schema to a corresponding variational query against a variational schema is defined in Section 3.4.

### 3.1 Variational sets and maps

A *variational set* $\{e_1^{c_1}, \ldots, e_n^{c_n}\}$ is a set of elements where each element $e_i$ is annotated by a *presence condition* $c_i$ [7, 16]. A presence condition is a Boolean formula of *configuration options* [9], where a configuration option is simply a Boolean variable that can be enabled (true) or disabled (false). The syntax of presence conditions is defined in Figure 1.

Conceptually, a *variational set* represents many different plain sets that can be generated by enabling or disabling all of the configuration options and including only the elements where the presence conditions evaluate to true. For example, the variational set $\{2^A, 3^B, 4^{A \vee B}\}$ represents four different plain sets: $\{2, 4\}$ if $A$ is enabled but $B$ is disabled, $\{3, 4\}$ if $B$ is enabled but $A$ is disabled, $\{2, 3, 4\}$ if both are enabled, and the empty set if both are disabled.

We indicate variational sets with an arrow, as in $\vec{S}$, as a reminder that variational sets are conceptually a function from a configuration of its options to the corresponding plain sets. We typically omit the presence condition `true` when writing variational sets, for example, in the variational set $\{5, 6^A\}$, the presence condition for the value 5 is implicitly `true`, and so the element is included in both variants of the set.

Similarly, a *variational map* $\vec{\vec{M}}$ (note the different arrow) represents many different plain maps. A variational map associates each key $k_i$ with a variational set of values $\vec{V_i}$, that is $\vec{\vec{M}}(k_i) = \vec{V_i}$. An invariant of the variational map representation is that the presence conditions in each $\vec{V_i}$ in the range of $\vec{\vec{M}}$ partition the configuration space; that is, every variant of $\vec{V_i}$ should have exactly one or zero elements. If a $k_i$ maps to the empty set in some configuration of $\vec{\vec{M}}$, then that variant of $\vec{\vec{M}}$ does not contain a mapping for $k_i$, otherwise $k_i$ is mapped to the element $v_i$ of the configured set $\vec{V_i}$. For example, consider the variational map $\{x \mapsto \{2^A, 3^{\neg A}\},\ y \mapsto \{4^A\}\}$, which represents two plain maps: $\{x \mapsto 2, y \mapsto 4\}$ when $A$ is enabled, and $\{x \mapsto 3\}$ when $A$ is disabled.

Schemas:

$$
\begin{array}{rcl}
R \in \mathit{Rel} \quad A \in \mathit{Attr} & & \\
s \in \mathit{Spec} & ::= & R(A_1, \ldots, A_n) \\
S \in \mathit{Schema} & ::= & \{s_1, \ldots, s_n\}
\end{array}
$$

Queries:

$$
\begin{array}{rcl}
k \in \mathit{Const} \quad x \in \mathit{Arg} & & \\
a \in \mathit{Arg} & ::= & k \mid x \mid \_ \\
f \in \mathit{Formula} & ::= & R(a_1, \ldots, a_n) \mid a_1 \bullet a_2 \\
\bullet \in \mathit{Op} & ::= & \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{!=} \\
g \in \mathit{Goal} & ::= & f \mid \texttt{not}\ f \\
r \in \mathit{Rule} & ::= & f\ \texttt{:-}\ g_1, \ldots, g_n \\
q \in \mathit{Query} & ::= & f\ \texttt{with}\ r_1, \ldots, r_n
\end{array}
$$

**Figure 2: Abstract syntax of Datalog.**

### 3.2 Variational schemas

We start with variational schemas. For reference, the abstract syntax of schemas and queries in plain Datalog is given in Figure 2. A schema is defined as a set of relation specifications, where each specification provides the name of the relation and an attribute name for each argument. Conceptually, a variational schema represents many different plain schemas that can be obtained by configuration.

A naive representation of variational schemas would be to make the set of 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 of a schema in which the attributes of a relation have been reordered. Expressing all such reorderings explicitly using a variational set of relation specifications is tedious and obfuscates more interesting variations. Instead, we encode relation specifications as (variational) sets of attributes rather than lists.

A trickier case is attribute renaming. Consider renaming the attribute $X$ to $Y$ of a relation $R$ based on a configuration option $A$. We might represent this using the following variational relation specification $R\ \{X^{\neg A}, Y^A, Z\}$, which also supports reordering $X$ or $Y$ with attribute $Z$ (present in all variants of the relation). However, it's not clear from this encoding that $X$ and $Y$ conceptually represent the same attribute, as opposed to two different attributes that are conditionally included in one variant or the other.

$$
\begin{array}{rcl}
f' \in \mathit{VFormula} & ::= & R\,\{i_1 : a_1^{c_1}, \ldots, i_n : a_n^{c_n}\} \;\mid\; a_1 \bullet a_2 \\
g' \in \mathit{Goal} & ::= & f' \;\mid\; \texttt{not}\; f' \\
r' \in \mathit{Rule} & ::= & f' \texttt{ :- } g_1'^{c_1}, \ldots, g_n'^{c_n} \\
q' \in \mathit{Query} & ::= & f' \texttt{ with } r_1'^{c_1}, \ldots, r_n'^{c_n}
\end{array}
$$

**Figure 3: Variational queries.**

To make the conditional renaming of attributes clear, we introduce a set *ID* of unique attribute identifiers, and refer to these in variational relation specifications. A variational schema then contains both the variational set of variational relation specifications, and a variational map from identifiers to attribute names. For example, we can represent the variational relation above with the specification $R\,\{1, 2\}$ and the variational map $\{1 \mapsto \{X^{\neg A}, Y^{A}\},\ 2 \mapsto \{Z\}\}$.

The representation of variational schemas can thus be summarized as a pair $(\overrightarrow{N}, \vec{S})$, where $\overrightarrow{N}(i) = \vec{A}$ for $i \in \mathit{ID}$ and $\vec{A}$ is a variational set of attribute names. The variational set $\vec{S}$ contains variational relation specifications of the form $R\,\vec{I}$, where $\vec{I}$ is a variational set of the identifiers included in relation $R$.

Assume variations in Table 1 only included tables *info*, *subjectInfo*, and *physician*. A variational schema that includes both variants is $(\overrightarrow{N}, \vec{S})$, where $\overrightarrow{N}$ and $\vec{S}$ are defined as below.

$$
\begin{array}{l}
\overrightarrow{N} = \{1 \mapsto \{\mathit{id}\},\ 2 \mapsto \{\mathit{gender}\},\ 3 \mapsto \{\mathit{date}\}\}, \\
\qquad 4 \mapsto \{\mathit{physician}^{E}, \mathit{name}^{\neg E}\}, \\
\vec{S} = \{\mathit{info}\{1,2,3,4\},\ \mathit{subjectInfo}^{\neg E}\{1,2\},\ \mathit{physician}^{\neg E}\{1,4,3\}\}
\end{array}
$$

The configuration option $E$ determines whether we are considering the first variation or the second one. Note that the variational schema does not capture the revised ordering in the evolved schema, but rather all possible ordering of the attributes.

## 3.3 Variational queries

Conceptually, a variational query describes a query that can be executed over any database consistent with the variational schema. The property that must hold between a variational query $q'$ and a variational schema $S'$ is that for every plain query $q_C$ obtained from $q'$ by configuring with a function $C : \mathit{Opt} \to \mathit{Bool}$, $q_C$ is consistent with the corresponding plain schema $S_C$ obtained by configuring $S'$ with the same function $C$. That is, every variant query matches the corresponding variant schema. In previous work, we have shown how similar consistency properties can be efficiently checked [3, 4].

The representation of variational queries in Figure 3 builds on the representation of plain queries in Figure 2. First, relations are generalized to accept a variational set of named arguments, where the names are the unique identifiers defined in the variational schema. Second, the sub-goals of a rule are annotated by presence conditions indicating in which variants those sub-goals are applicable. Third, each rule in a query is itself annotated by a presence condition indicating in which variants the rule should be included.

## 3.4 Lifting plain queries to variational queries

One goal of our approach is to support writing a plain query over an example database, then generalize or lift that query using a variational schema to work on variant layouts of that database. Since variational schemas can express arbitrary variability between schemas (e.g. a degenerate case is a variational schema that encodes two completely different and unrelated schemas), it is not in general possible for this lifting process to produce a semantically equivalent query for all variants.

However, lifting should produce a variational query that has the following two properties: (1) it is consistent with the variational schema as defined in Section 3.3, and (2) the variant queries should be semantically equivalent for all variant schemas that different from the initial schema only by attribute renaming or reordering.

We write $\lceil f \rceil_S^{S'}$ to represent lifting a plain formula $f$ defined over schema $S$ to a variational formula $f'$ defined over a variational schema $S'$. The definition of formula lifting is given below. In the definition, $S(R, j) = A_j$ is the attribute name $A_j$ for the $j$th argument of relation $R$ in $S$; $S'(R, A_j) = i^c$ is the identifier and presence condition for attribute $A_j$ of relation $R$ in $S'$; and $S'(R) = \vec{I}$ is the variational set of identifiers associated with relation $R$ in $S'$.

$$
\begin{array}{rl}
\lceil R(a_1, \ldots, a_n) \rceil_S^{S'} & = R\,\{i : \_^{c} \mid i^c \in S'(R),\ i \notin I\} \\
& \quad \cup \{i : a_j^c \mid j \in \{1..n\},\ i^c = S'(R, S(R, j))\} \\
\lceil a_1 \bullet a_2 \rceil_S^{S'} & = a_1 \bullet a_2
\end{array}
$$

The lifted formula essentially augments the initial formula with the unique identifiers and presence conditions corresponding to the given arguments, then pads out the relation with arguments that match anything (_) for all attributes that exist only in other variants of the schema.

With the function for lifting formulas, lifting goals, rules, and queries is straightforward. For goals, we simply lift the argument formula, for rules we lift each goal setting the presence conditions to `true`, and likewise for queries we lift each rule setting the presence conditions to `true`.

Assume designers in Example 1.1 want to get a list of subjects and their physicians. While they need to query the first variation with: $A(x, y) :- \mathit{info}(x, a, b, c, d, y)$, they need to query the second variation in Table 1 with $A'(x, y) :- \mathit{subjectInfo}(x, a, b)$, $\mathit{physician}(x, y, c)$. These queries can be lifted to the variational schema defined in Section 3.2 to produce the following variational query:

$$
\begin{array}{l}
A\,\{1 : x, 4 : y\} \texttt{ :- } \mathit{info}\,\{1 : x, 2 : a, 3 : b, 4 : y\}, \\
\qquad \mathit{subjectInfo}^{\neg E}\,\{1 : x, 2 : a\},\ \mathit{physician}^{\neg E}\,\{1 : x, 4 : y, 3 : b\}
\end{array}
$$

To obtain the plain query for the second variation $S_E$, we first configure this variational query with $E = \texttt{false}$, then substitute the IDs for the corresponding attribute names, and finally order the arguments according $S_E$ and remove the attribute names.

After lifting, a variational query can be manually augmented to support other variants described by the schema, which differ from the initial schema arbitrary ways. This can be done by editing the variational query directly, or by projecting on the relevant variant(s) of the query and editing this simplified view [12, 17].

# 4 CHALLENGES

In this section we discuss challenges that we foresee in using variational schemas and variational queries in practice.

**Generalizing queries over other systematic transformations:** The variational schema representation immediately supports several kinds of refactorings, including renaming, reordering, adding, and removing attributes. Other kinds of refactorings, such as renaming relations, would be easy to add using similar techniques to those we have already applied. However its not clear how

best to support other kinds of refactorings, such as (de)composition of relations. The variational schema representation supports arbitrary variation among schemas by conditionally including or not different variants of the same relation, however, encoding refactorings using these techniques means that we lose more specific information about the refactoring that was applied and so cannot automatically generalize the query to work on such variants. Therefore, future work should consider how to encode other kinds of transformations in variational schemas.

**Ensuring properties of generalized queries:** A benefit of the variational schema representation is that it can express arbitrary variation among schemas, and is not limited only to a set of standard schema refactorings. However, this expressiveness limits the strength of guarantees we can make about the lifting process. As described in Section 3.4, lifting ensures only that a variational query is consistent with the variational schema in the sense that each variant query uses relations that are defined and of the appropriate arity. Building on our previous work on variational typing [3, 4], we could also guarantee that arguments in each variant query are all of the correct type. In the case of simple refactorings like renaming and reordering variants, it should be easy to prove that lifting also preserves the semantics of the query across all variant schemas. However, for many other kinds of variation in schemas, lifting will not ensure semantic equivalence across variants and will often require manually augmenting the query to support variants not covered by the original query. So what can we say about such variational queries? Future work should consider how to express and prove properties about variational queries that vary in arbitrary ways. Existing work on variational analysis (e.g. in the context of software product lines [14]) can provide guidance here.

## 5 RELATED WORK

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 [13]. 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.

Dataset versioning systems manage multiple versions of a dataset created as a result of database updates over time [2]. They enable users to query certain versions of a dataset. Nevertheless, users of variational databases would like to have a variation-independent view of the data and query all variations simultaneously. Variational databases also seek to model general forms of data variations, e.g., representational variations. Schema independent data analytics systems return semantically equivalent results over different schematic variations of the same dataset [10]. Variations of a variational database, however, may have different amount of information.

The representation of variational queries differs from our previous approach to encoding variation in abstract syntax trees [6, 15] 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`. The representation chosen here, based on presence conditions, is similar to our previous work on variational lists, sets, and graphs [7, 16]. An advantage of choice-based variation is that we can apply a set of laws for factoring and distributing choices within the abstract syntax tree, which can be used to minimize redundancy in the representation. However, the representation chosen here is a better fit for variation in collections (e.g. sets of arguments, lists of sub-goals and rules) [16].

## 6 CONCLUSION AND FUTURE WORK

We introduced variational schemas and variational queries to address the problem of query dependence on a particular representation of data. Note that although we chose relational databases and Datalog queries, this problem and this approach are applicable to other data sources, such as spreadsheets, and other query languages, such as SQL. As future work, we will address the challenges described in Appendix. We will implement our approach to study it empirically. And we will extend our model of variational databases to support a wider variety of systematic transformations and other kinds of data sources, such as spreadsheets. The concept of variational databases can be used in a variety of applications including ,but not limited to, database versioning, schema evolution, relational learning algorithms, feature selection, and software maintenance.

## REFERENCES


[1] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer-Verlag, Berlin, 2016.

[2] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow.*, 8(12):1346–1357, Aug. 2015.

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

[4] S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 36(1):1:1–1:54, 2014.

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

[6] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 21(1):6:1–6:27, 2011.

[7] M. Erwig, E. Walkingshaw, and S. Chen. An Abstract Representation of Variational Graphs. In *Int. Work. on Feature-Oriented Software Development (FOSD)*, pages 25–32, 2013.

[8] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. Parameswaran. Orpheusdb: Bolt-on versioning for relational databases. *Proc. VLDB Endow.*, 10(10):1130–1141, June 2017.

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

[10] J. Picado, A. Termehchy, A. Fern, and P. 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.

[11] M. Stonebraker, D. Deng, and M. L. Brodie. Database decay and how to avoid it. In *Big Data (Big Data), 2016 IEEE International Conference*. IEEE, 2016.

[12] S. Stănciulescu, T. Berger, E. Walkingshaw, and A. Wasowski. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, pages 323–333, 2016.

[13] D. Suciu, D. Olteanu, R. Christopher, and C. Koch. *Probabilistic Databases*. Morgan & Claypool Publishers, 1st edition, 2011.

[14] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)*, 47(1):6, 2014.

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

[16] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. 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.

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