Managing Structurally Heterogeneous Databases in Software Product Lines

Parisa Ataei, Arash Termehchy, and Eric Walkingshaw

Oregon State University, Corvallis OR 97331, USA
{ataeip,termehca,walkiner}@oregonstate.edu

Abstract. Data variations are prevalent while developing software product lines (SPLs). A SPL enables a software vendor to quickly produce different variants of their software tailored to variations in their clients' business requirements, conventions, desired feature sets, and deployment environments. In database-backed software, the database of each variant may have a different schema and content, giving rise to numerous data variants. Users often need to query and/or analyze all variants in a SPL simultaneously. For example, a software vendor wants to perform common tests or inquiries over all variants. Unfortunately, there is no systematic approach to managing and querying data variations and users have to use their intuition to perform such tasks, often resorting to repeating a task for each variant. We introduce *VDBMS* (Variational Database Management System), a system that provides a compact, expressive, and structured representation of variation in relational databases. In contrast to data integration systems that provide a unified representation for all data sources, VDBMS makes variations explicit in both the schema and query. Although variations can make VDBMS queries more complex than plain queries, a strong static type system ensures that all variants of the query are consistent with the corresponding variants of the database. Additionally, *variational queries* make it possible to compactly represent and efficiently run queries over a huge range of data variations in a single query. This directly supports many tasks that would otherwise be intractable in highly variational database-backed SPLs.

Keywords: Variational databases · Variational queries · VSQL · Variational relational algebra · Software product lines · Heterogeneous databases

1 Introduction

Data variation is ubiquitous when developing software. Each domain contains numerous databases which differ in terms of schema, data representation, and/or content. In fact, even a single software vendor or project may need to maintain many different data representations of the same concepts. One way this arises is in the context of a *software product line* (SPL), which is a single software project that can be used to generate many different program variants [1]. For instance, software vendors often customize their software and create distinct variants for each client based on the client's geographical settings, business requirements, and

Feature	Associated relation
-	$message(ID, sender, date, subject, body)$
-	$recepientInfo(rid, ID, rtype)$
Encryption	$encryption(ID, isEncrypted, encryptionKey)$
Signature	$signature(ID, signed, signKey)$
Verification	$verification(ID, isVerified)$
US	$person(ID, firstName, middleName, lastName)$
France	$person(ID, firstName, lastName)$
Iceland	$person(ID, firstName, fatherName, gender)$

Table 1. A subset of features in an email system SPL and the relations associated with them. If a feature is enabled, the schema of the email database includes the corresponding relation. The first two relations are included in all variants of the SPL. The three different *employee* relations, associated with different countries' naming conventions, are mutually exclusive.

the capabilities it wants. Most open source projects have hundreds or thousands of static configuration options, yielding a staggering number of variants [8].

Different variants of a SPL require different data representations. Consider a vendor that develops an email messaging system for customers around the globe. Each country may have a different standard of naming people, for example, in contrast to the US, there is no notion of a *middle name* in France, and in Iceland, a person's last name is determined by their gender and their father's name. Hence, this vendor may have to create a distinct relation schema for the relation *person* according to the country of the customer as shown in Table 1. A common intent in this email system is to retrieve a person's *ID* by their full name. To express this intent, a developer of the system has to write a distinct SQL query for each naming convention in their code. This problem grows multiplicatively when different variations interact, for example, a query to retrieve full names combined with an optional privacy feature, may require two distinct queries for each naming convention. Thus, a developer may end up writing many SQL queries to express the same intent across many software variants.

The database community has long recognized that users must modify their queries to preserve their semantic and syntactic correctness over various schemas and has proposed (declarative) schema mappings to solve this problem [3,5]. For example, in the context of schema evolution, one first defines or discovers the mapping between the original and new schemas. Given this schema mapping, one can safely and automatically translate the queries written over the original schema to the new one. Of course, the new schema must contain the information the query needs. However, the variations in a SPL do *not* enjoy this property. For instance, if one knows the mapping between the schema for the client in the US and the one in France, they cannot automatically translate the query written for the France-based client to the one for the US-based client as the schema mapping does not imply the need to use the attribute middle name to preserve the intent behind the query. In fact, if one follows the mapping between the schemas, the query written for the US-based client will not use the attribute *middle name*!

In Section 2, we describe how such variation is managed in real-world SPLs and why current approaches are unsatisfactory to developers.

As a solution, we propose *VDBMS* (Variational Database Management System), a system that manages structurally heterogeneous databases in similar context and allows users to query multiple variants simultaneously without losing data provenance. In Section 3, we describe the core concepts of VDBMS, and describe the architecture of the VDBMS system in Section 4.

2 Motivating Example

One way of defining a SPL is to identify and model the *features* that give rise to different variants of the software [1]. For our purposes, a feature is a name that corresponds to some potentially optional unit of functionality, and a *feature model* describes the relationship between features. In the example illustrated in Table 1, *US*, *France*, and *Iceland* are features corresponding to different naming conventions and are mutually exclusive according to the feature model (not shown in the table). Other features in the email system include *Encryption*, *Signature*, and *Verification*. If, say, the *Encryption* feature is enabled, then the corresponding software variants will encrypt and decrypt emails. Features can be combined in different ways and extend or modify a shared code base that implements the basic requirements of the system shared across all variants, such as sending and receiving messages in an email system. By organizing the variability of a SPL around features, a vendor or project can share significant costs and effort in developing and maintaining many software variants [1].

Generating, managing, and maintaining separate schemas for each variant in a SPL is not simply tedious, but often impossible since the number of variants grows exponentially with the number of independent features. From our conversation with SPL experts, the dominant workaround is to create a global schema that contains all relations and attributes used across all variants of the software, then write queries over this global schema. However, such a schema may not be meaningful. For instance, in our email system example, the global schema must contain all attributes required to store various naming conventions for the relation *person*, which will not have any instance in the real world. Also, numerous tuples in the database will contain null values, e.g. middle names for all people in France. If the database is deployed and resides on the client's location, the client has a large schema but uses only a small subset of it. Developers must write distinct SQL queries for different software variants to express an intent that is shared among all variants. It may also be error-prone to write a query directly over such a global schema, as the query has access to many attributes and relations that do not make sense in its variants.

A cleaner approach is to define a view over the global schema for each variant and write queries for each variant against its view. However, developers then have to generate and maintain numerous view definitions and must still write many SQL queries to express the same intent. The developer must manually generate and manage the mappings between views and the global schema for each client.

As a result, while querying database variants, they face similar problems to ones mentioned for schema mapping methods in Section 1. Moreover, update queries after deployment must deal with the problems of view-updating since the base tables of the products are defined as views. This approach works for a SPL with a small number of clients/variants. However, it doesn't scale to open-source SPLs where the selection of an individual variant is up to the end-user, and the space of potential variants is massive.

VDBMS introduces a novel abstraction called a *variational schema*, a compact representation of all schemas used by the software variants of a SPL, where the presence of relations and attributes in the schema is defined in terms of the features of the SPL. It also provides a novel *variational query* language that enables SPL programmers to refer to features explicitly. Instead of writing separate queries for each variant of a SPL, programmers can express an intent over all possible schema variants of a SPL in a single query. By making variation explicit in schema and queries, VDBMS simplifies the task of testing and maintaining database-related functionality across software variants. Finally, it provides opportunities for sharing query processing across multiple schema variants.

3 Variational Database Framework

A *variational database* (VDB) is conceptually a set of relational database variants that may each have a different schema. It is conceptually useful in any context where one wants to work on some/all of these variants simultaneously.

3.1 Variational Schema

Similar to relational databases, we need to *compactly* express the schema of a VDB. We assume that different variants of a SPL can either include or exclude a relation, and if they include a relation they can either include or exclude an attribute of that relation. A *variational schema* (v-schema) concisely encodes the plain relational database schemas for all of the software variants in a SPL [2]. The representation of v-schemas is based on the formula choice calculus [4,7].

Conceptually, a variational schema is just a relational schema with embedded *choices* that locally capture the differences among variants. A choice $F\langle x, y\rangle$ consists of a *feature expression* F and two alternatives x and y. A feature expression is a propositional formula over the *features* of the SPL, where each feature can either be enabled (*true*) or disabled (*false*). For a particular set of enabled features, the choice $F\langle x, y\rangle$ can be replaced by x if F evaluates to true, or y otherwise. Each software variant of the SPL corresponds to a set of enabled features (its *configuration*); the plain schema for that variant can be obtained by simply eliminating each of the choices in the v-schema as described above.

A v-schema allows for the embedding of choices within the sets of attribute names, forming *variational relation schemas*. We illustrate this in Example 1.

Example 1. Assume our schema contains the relation *person* and our SPL contains the country-specific features. Then $A = \mathit{US}\langle l_1 \cup \{\mathit{middleName}\}, \mathit{France}\langle l_1, l_2\rangle\rangle$

encodes the set of attribute names for the *person* relation shown in Table 1, where $l_1 = \{ID, firstName, lastName\}, l_2 = \{ID, firstName, fatherName, gender\}$. Note that l_2 contains the attributes for the *Iceland* feature, which are included when neither *US* nor *France* are enabled. The entire v-schema can be represented as $S = (US \vee France \vee Iceland)\langle person(A), \varnothing\rangle$, where $person(A)$ is a variational relation schema (v-relation schema) and $\varnothing$ indicates a non-existing schema.

A *v-relation* is a set of tuples that conform to the same v-relation schema, where each tuple has a feature expression that indicates the software variants that include the tuple (its *presence condition*). A set of v-relations form a VDB.

Within a SPL, not all configurations yield valid software variants. For example, any valid configuration of our email system contains *exactly one* of the features *US*, *France*, and *Iceland*. In practice, the set of valid configurations of a SPL is described by a *feature model* [1]. Here, we consider a feature model to be a feature expression that is satisfied iff the configuration is valid. For example, the corresponding fragment of our email system feature model is: $(US \wedge \neg France \wedge \neg Iceland) \vee (\neg US \wedge France \wedge \neg Iceland) \vee (\neg US \wedge \neg France \wedge Iceland)$

The feature model is an input to VDBMS and is implicitly applied globally. For example, given the feature model above, the nested choice $US\langle x, France\langle y, z\rangle\rangle$ will resolve to x for the US, y for France, and z for Iceland. Although for simplicity we use propositional formulas for feature expressions and feature models, our model can be easily generalized to other encodings, such as first-order logic.

3.2 Variational SQL

To query a VDB, we introduce the notion of a *variational query* (v-query), which returns a v-relation. We define *variational SQL* (VSQL) as an extension of SQL with a new function `CHOICE`(f, e_1, e_2), where f is a feature expression and e_1 and e_2 may be VSQL queries, attribute sets used in a `SELECT` clause, relations (or joins of some relations) used in a `FROM` clause, or conditions in a `WHERE` clause. With VSQL, the SPL developer can use the `CHOICE` function to indicate different attributes, conditions, and relations to use for different variants of the database. This enables expressing a single intent across a potentially huge number of configurations in a single v-query.

In our examples, we use *variational relational algebra* (VRA) rather than VSQL, for brevity. VRA is relational algebra extended by the choice notation introduced in Section 3.1. However, we expect end-users to prefer the VSQL notation. Example 2 illustrates different ways of writing a v-query in VRA.

Example 2. Consider again the schema sketched in Table 1 and defined in Example 1. Suppose a developer would like to express the intent of querying a last name by projecting the last name attribute for US and France, and the father's name and gender in Iceland. They can do this with the following query: $Q_1 = \pi_{(France \vee US)\langle\{lastName\},\{fatherName, gender\}\rangle} person$. The output of this query is a v-relation that has the *lastName* attribute for both US and France variants and the *fatherName* and *gender* attributes for the Iceland variant of the data. The user may also submit the following query with nested choice expression to articulate the same intent: $Q'_1 = \pi_{\mathit{France}\langle\{\mathit{lastName}\},(\mathit{US}\langle\{\mathit{lastName}\},\{\mathit{fatherName},\mathit{gender}\}\rangle)\rangle}\mathit{person}$. Without choices and a VDB, expressing this query requires executing two different plain queries against three different databases.

Since VSQL is a strict superset of SQL, a developer may still write queries in plain SQL when the intent is expressed the same way across all variants. That is, VSQL does not impose additional complexity when it is not needed. Additionally, we employ type inference and a strong static type system that enables omitting choices in many cases where the variation in the v-query is completely determined by the corresponding variation in the v-schema. For example, we can express query Q_1 in Example 2 more simply as $\pi_{\mathit{lastName},\mathit{fatherName},\mathit{gender}}\,\mathit{person}$. This will project the *lastName* attribute for the US and France, and the *fatherName* and *gender* attributes for Iceland, and the inferred type of this query will track which attributes and tuples are present in which variants.

The type system also supports usability by ruling out invalid v-queries. For example, a query that contains the condition $\mathit{lastName} = \mathit{fatherName}$ would be invalid since there is no configuration of the database that includes both the *lastName* and *fatherName* attributes of the *person* relation.

Type inference enables omitting the "boring" choices that would only be needed to ensure consistency between the v-query and the v-schema, which in turn ensures that each variant of the v-query is structurally consistent with its corresponding variant of the VDB. This frees choices to be reserved for the more interesting cases where a v-query must describe *unsystematic* or *non-structurally determined* differences amongst its variants. This is illustrated in Example 3.

Example 3. Suppose we want to read the body of all emails. Our query must take into account whether an email is encrypted or not. This is illustrated by the following query with a choice over the feature *Encryption*, where Δ is a user-defined function that takes attributes *encryptionKey* and *body* and decodes the body according to the key.

$$\begin{aligned}&\mathit{Encryption}\langle\pi_{\Delta(\mathit{body},\mathit{encryptionKey})}(\sigma_{\mathit{isEncrypted}=\texttt{true}}\mathit{message} \bowtie_{\mathit{ID}=\mathit{ID}} \mathit{encryption})\\ &\qquad\cup\ \pi_{\mathit{body}}(\sigma_{\mathit{isEncrypted}=\texttt{false}}\mathit{message} \bowtie_{\mathit{ID}=\mathit{ID}} \mathit{encryption}), \pi_{\mathit{body}}\mathit{message}\rangle\end{aligned}$$

Note that variation in this query is not determined by the v-schema since each alternative of the choice not only queries different attributes and relations, but must also perform different functionality (namely, decoding the email body).

4 VDBMS Architecture

Fig. 1 shows the architecture of VDBMS. V-schema and v-query are supported by the VDBMS abstraction layer to enable the SPL developer to interact with the VDB. The SPL developer can include v-queries in the SPL codebase or input them to VDBMS directly. In this section, we briefly report our ongoing effort of implementing VDBMS using an existing RDBMS.

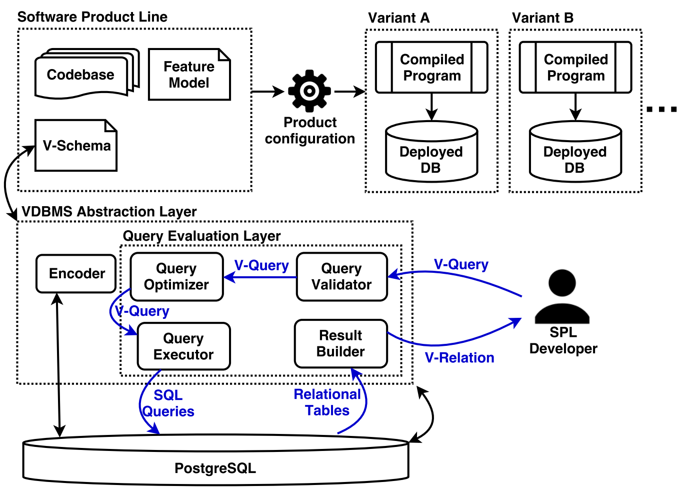

Fig. 1. Overview of VDBMS. The v-schema captures variation in database layouts and is accessible from all modules within the VDBMS layer.

4.1 Encoding the Variational Database

We implement VDBMS on top of PostgreSQL. All variants of a v-relation are encoded as a single relational table in PostgreSQL. This table contains the union of all attributes contained in all variants of the relation schema. We encode both the v-schema and the feature model as additional tables in PostgreSQL. The v-schema associates with each attribute a feature expression, called the *presence condition*, indicating in which variants the attribute is included.

A key aspect of a VDB is that it conceptually represents many different variant databases, and often it is important to both keep these databases distinct, and to keep track of which results come from which databases. One scenario where this is especially important is when features correspond to different clients, in which case we want to ensure that data associated with different clients do not mix. Therefore, our system must not only manage structural differences between variants, but also track which *data* is associated with which variants. We do this by attaching a presence condition to each tuple that indicates the variants in which the tuple is present. The presence condition is represented as a feature expression that VDBMS maintains and updates throughout the execution of a v-query. The key property enforced by maintaining presence conditions on tuples is *variation preservation*, which states that running a v-query on a v-database yields a v-relation that is equivalent to running each variant of the v-query on the corresponding variant of the v-database.

4.2 Optimizing and Evaluating Variational Query

The evaluation of a v-query proceeds in several steps. First, the *query validator* applies the rules of the type system to check whether the query is consistent with the variational schema. For example, the query $\pi_{(France\langle middleName, \varnothing\rangle)} person$ is invalid since it projects the *middleName* attribute when the *France* feature is enabled, but this attribute is present only when the *US* feature is enabled.

The *query optimizer* translates the v-query into a tree whose internal nodes are either relational operators or choices, and whose leaves are v-relations. The optimizer then applies equivalence laws from relational algebra and the choice calculus to achieve better performance.

Conceptually, the *query executor* executes an optimized variational query by translating it into a sequence of relational query operations interspersed by operations that enforce the variation-preservation property (Section 4.1). In practice, this is achieved efficiently by embedding user-defined functions in queries that can be executed entirely with the PostgreSQL DBMS. During the execution of a variational query, tuples can be filtered out of intermediate results not only by the selection predicate, but also because the tuples are not present in the variations that are applicable to that part of the query. Additionally, the presence conditions of tuples will be refined as they are processed by the query.

Finally, VDBMS must return a v-relation to the user. The *result builder* module collects the results, including the presence conditions of the relation, attributes, and tuples, and assembles them into a v-relation to return. Note that the query executor and result builder modules can work in a pipeline since the tuples' presence conditions are independent from one another.

5 Related Work

OrpheusDB supports *database versioning* [6]. Both OrpheusDB and VDBMS provide access to some versions or variants of a database at a time. However, unlike database versioning, which manages heterogeneity of content only, VDBMS also supports heterogeneous structure, that is, different schemas for different variants. Both database version and VDBMS support data sharing among versions/variants. In VDBMS, this is supported by presence conditions on tuples that are consistent with many different configurations. For example, a tuple with presence condition $US \vee France$ is included in all variants with either the *US* or *France* feature enabled (regardless of the configuration of other features).

Multi-tenant databases [9] take an architectural approach towards sharing resources among various organization that use different applications and hence different databases without any limitation on database variations. They do so by storing data ownership and the database schema in relational tables. However, VDBMS is only used for databases in similar contexts since it adds a level of abstraction to both the schema and content of the database. As a result, it allows for as much sharing as possible among database variants while multi-tenant databases do not allow for any sharing since the variations can be completely different. Interestingly, they both secure client's information, VDBMS does so by providing the *variation-preservation* property and multi-tenant databases do so by tagging the *client ID* to data.

6 Conclusion and Future Work

While developing SPLs, developers must deal with many variants of a database corresponding to different configurations of the software. Maintaining each database and its corresponding set of queries manually doesn't scale to highly configurable SPLs. Alternative solutions, such as including all of the information for all variants in a single schema, are error-prone and don't address the problem of unsystematic variation, that is, when different configurations of the software may require different queries to express the same intent, which are not determined by differences in structure alone. We introduced a conceptual framework for VDBMS, including v-schemas, which compactly represents the schema associated with each configuration of an SPL, and variational queries, which enable users to express both systematic and unsystematic variations of a single intent across all variants of the database. We have also introduced the VDBMS architecture, including how it integrates with the SPL and how it is realized in the underlying DBMS, PostgreSQL. VDBMS enforces a variation-preservation property that ensures that queries and data associated with different configurations remain distinct and consistent.

We plan to extend VDBMS to allow for disciplined overriding of the variation-preservation property, to enable combining results from many different variants in a single v-query. We also plan to explore further optimizations to the system to improve performance, and how to extend VDBMS to support other use cases besides SPL development.

References

1. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer-Verlag, Berlin (2016)
2. Ataei, P., Termehchy, A., Walkingshaw, E.: Variational Databases. In: Int. Sym. on Database Programming Languages (DBPL). pp. 11:1–11:4 (2017)
3. Doan, A., Halevy, A., Ives, Z.: Principles of Data Integration. Morgan Kaufmann, San Francisco (2012)
4. Erwig, M., Walkingshaw, E.: The Choice Calculus: A Representation for Software Variation. ACM Trans. on Software Engineering and Methodology (TOSEM) **21**(1), 6:1–6:27 (2011)
5. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data Exchange: Semantics and Query Answering. In: Int. Conf. on Database Theory (ICDT) (2003)
6. Huang, S., Xu, L., Liu, J., Elmore, A.J., Parameswaran, A.: OrpheusDB: Bolt-on Versioning for Relational Databases. Proc. of the VLDB Endowment **10**(10), 1130–1141 (Jun 2017)
7. Hubbard, S., Walkingshaw, E.: Formula Choice Calculus. In: Int. Work. on Feature-Oriented Software Development (FOSD). pp. 49–57 (2016)
8. Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M.: An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In: ACM/IEEE Int. Conf. on Software Engineering. pp. 105–114 (2010)
9. Weissman, C.D., Bobrowski, S.: The Design of the Force.com Multitenant Internet Application Development Platform. In: ACM SIGMOD Int. Conf. on Management of Data (SIGMOD). pp. 889–896 (2009)