SPL Driven Approach for Variability in Database Design

Selma Bouarar[1], Ladjel Bellatreche[1], Stéphane Jean[1], and Norbert Siegmund[2]

[1] LIAS/ISAE-ENSMA - Poitiers University, France
(bouarars, bellatreche, jean)@ensma.fr
[2] Department of Informatics and Mathematics, University of Passau
Norbert.Siegmund@uni-passau.de

Abstract. The evolution of computer technology has strongly impacted the $\mathcal{DB}$ design. No phase was spared: several *conceptual* formalisms (e.g. ER, UML, ontological), various *logical* models (e.g. relational, object, key-value), a wide panoply of *physical* optimization structures and *deployment* platforms have been proposed. As a result, the $\mathcal{DB}$ design process has become more complex involving more tasks and even more actors (as $\mathcal{DB}$ architect or analyst). Getting inspired from software engineering in dealing with variable similar systems, we propose a methodological framework for a variability-aware design of $\mathcal{DB}$, whereby this latter is henceforth devised as a Software Product Line. Doing so guarantees a high reuse, automation, and customizability in generating ready-to-be implemented $\mathcal{DB}$. We also propose a solution to help users make a suitable choice among the wide panoply. Finally, a case study is presented.

Keywords: Database design, Software Product Line, Variability.

1 Introduction

The proliferation of information systems (e.g. decisional, statistical, and scientific) has led to the development of different systems for storing data. These latter are called databases ($\mathcal{DB}$), and they have been becoming increasingly fundamental for every sector. Once the $\mathcal{DB}$ technology became mature, a design life-cycle has emerged. Its definition has undergone several evolutionary stages before being accepted as it stands actually, i.e. mainly composed of three main phases: *conceptual*, *logical* and *physical* designs.

The growing use of $\mathcal{DB}$ and the parallel upward diversity in nowadays applications, and requirements, more powered by the *big data era*, have revolutionized the $\mathcal{DB}$ technology. These facts (i) have subjected the design life-cycle to a continuous evolution: heterogeneity of data sources, diversity of conceptual/logical models, storage layouts, wide panoply of optimization structures and even the integration of new phases such as the *ETL* phase, (ii) have led to new careers related to $\mathcal{DB}$ management besides the usual designer and DBA, like $\mathcal{DB}$ architect, analyst, and developer. Every actor has his own variables (sphere of operation) through which he can configure the design. A such high degree of variety and collaboration shows the increasing complexity of nowadays $\mathcal{DB}$ design process.

This observation has motivated us to closely analyze the latter and spot the main variables that control the life-cycle as well as their dependencies.

Originally, the idea of developing similar complex products belonging to a specific family, and based on a set of variables (assets) is a well known process in Software Product Line (SPL) Engineering, as *Variability management*. It is defined as the ability of a product or artifact to be changed, customized or configured for use in a particular context. It has proved very successful in many domains (e.g. avionic and medical systems), and thanks to several reasons, such as: re-usability and flexibility, reducing costs and time to market, improving productivity, product maintainability, quality and tailoring products so as to meet, as specific as possible, individual customers. Our proposal can be summed up in how to adopt the SPL approach in managing variability in $\mathcal{DB}$ technology.

By exploring the major state-of-art, we figured out that this issue has been partially addressed (i) using different techniques inspired from software engineering, (ii) concerned with only parts of separate phases of $\mathcal{DB}$ design process, despite their interdependence. For instance, Rosenmuller et al [12] proposed *FAME-DBMS*, a prototype of a $\mathcal{DB}$ management product line, with focus only on a part of the physical layer of the design process, but leaving out other important phases in the life-cycle, such as conceptual and logical ones. In a parallel line of research, they developed means to customize SQL according to the usage scenario [13] and to tailor the conceptual scheme of a $\mathcal{DB}$ according to the application [15]. However, these approaches present isolated solutions indicating even more the need for a holistic variability-aware development of $\mathcal{DB}$ systems. In this paper, we propose an *SPL-inspired* methodological framework for a variability-aware design of $\mathcal{DB}$. The framework allows developers to both (i) derive ready-to-be-implemented $\mathcal{DB}$ applications, by composing features related to the whole design of $\mathcal{DB}$, and (ii) evaluate the design process so as to help users make better choices. Our paper is structured as follows: Section 2 shows a review on the SPL approach. Section 3 presents our SPL-inspired methodology to manage the variability of $\mathcal{DB}$ design. Section 4 presents its valuing process. Some experiments are conducted in Section 5. Section 6 surveys existing research related to *variability* and the current vision of $\mathcal{DB}$ research community on it.

2 Background: Software Product Line Engineering

Initially, every software product was individually set up for a specific purpose. Quickly, such an approach became costly because developers invented the wheel again and again introducing the same bugs. Thus, software engineers had to focus on domain knowledge to make a more strategic reuse and an efficient mass customization. This results in a number of different approaches such as *SPL*, which has proved very successful in building families of similar systems. An SPL is a set of products that share significant common functionality and structure. As such, building each product is based on composing it from the common artifacts, as for the uncommon aspects, they are left open. We refer to these open spots as variability points. The development of an SPL involves two main processes: variability capture (identification & constraining), and variability implementation (product configuration & derivation) as further depicted in Fig.4.

i)- Identification of Variability:
Variability can be more easily identified if the system is modeled using the concept of features: "*a logical unit of behavior that is specified by a set of functional and quality requirements*"[2]. Fig.1 outlines what is a feature diagram.

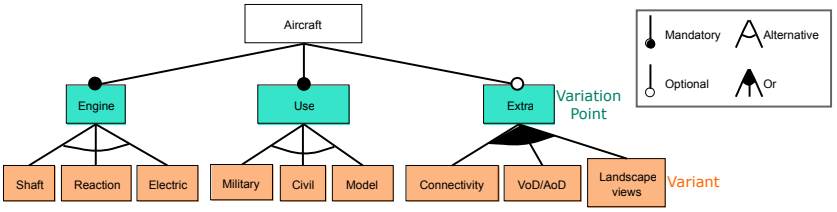

Fig. 1. Aircraft product line example.

- Variation point is a relevant characteristic (feature) of the PL, that may vary from an instance (product) to another. *Engine* is a mandatory unlike the *Extra* one which is optional.
- Variant is a realization (instantiation) of a feature. In other words, an alternative to bind the variation point. *Electric Engine* is one possible realization, among others, of the *Engine* feature.
- Scope of PL: description of the set of products which are parts of an SPL.

ii)- Constraining Variability: once features have been identified, not all possible combinations of them correspond to feasible SPL instances. To define the valid ones, constraints like *require* and *exclude* are added. E.g. *Military Use* excludes *Extra* feature, and *Civil Use* requires *Electric Engine*. A feature diagram must be enriched by constraints to turn into a feature model.

iii)- Implementing Variability: based on the previous steps, a suitable variability realization technique may be selected to define how the product derivation process is implemented. There exist a lot of platforms for feature-oriented software development like S.P.L.O.T and FeatureIDE[5].

In order to adopt the SPL approach in our context, we will follow these above steps as detailed in the next section.

3 $\mathcal{DB}$ Design as an SPL

Our first contribution aims at defining a variability-aware methodology, that devises $\mathcal{DB}$ design process as an SPL, and hence taking full advantage of this approach. Dealing with the $\mathcal{DB}$ design process as a whole allows users to have an overall vision, consider life-cycle interdependencies, and tackle all $\mathcal{DB}$ design steps while increasing process automation. Moreover, in contrast to the classic DBMS-to-$\mathcal{DB}$ vision, according to which, comes the DBMS selection before the logical step, we will see that our SPL-based approach can add more independence by delaying the DBMS selection to a later point in the process.

On the other hand, our modeling is far from being unique or exhaustive. However, we believe that we have defined the basic level of granularity, that

is sufficient to reach our objectives, namely to perform and evaluate the performance of the design process. Moreover, users can easily extend our feature models to additional needs. We first discuss the SPL framework that we use.

3.1 Generic SPL Framework

Our SPL is modeled as a feature model, defined by the couple: $\mathcal{FM} =< \mathcal{F}, \mathcal{IC} >$, such as $\mathcal{F} = \{f_1, ..., f_n\}$ is a set of features, and $\mathcal{IC}$ a set of integrity constraints (propositional formulas) over the set of features having the following form: $\mathcal{IC}_i =< f_1, exclude|require|recommend|recommend\text{-}not, f_2 >$.
Each feature is defined by a triplet: $\forall f \in \mathcal{F}, f =< t, \mathcal{FG}, \mathcal{I} >$ such as:

1- Each feature can be a parent of multiple feature groups $\mathcal{FG}$ (variants, children), such that $\mathcal{FG} =< \mathcal{FG}_{and}|\mathcal{FG}_{or}|\mathcal{FG}_{alt}|f|\varnothing >$, with:

- $\mathcal{FG}_{and} = (f_1 \wedge f_2)^+$, meaning that all mandatory children have to be selected whenever this parent is selected. It has no particular model notation.
- $\mathcal{FG}_{or} = (f_1 \vee f_2)^+$: at least one feature must be selected
- $\mathcal{FG}_{alt} = (f_1 xor f_2)^+$: exactly one of these children has to be selected
- If $\mathcal{FG} = \varnothing$, then f is a variant (a leaf node), else, it is a variation point.

2- t is the type of the feature among mandatory, optional, or undefined if the feature belongs to an $\mathcal{FG}_{or}$ or $\mathcal{FG}_{alt}$ groups.

3- $\mathcal{I}$ is the input required for implementing the feature, since our features are binary code. We will instantiate the framework as we move forward.

3.2 Identifying Variability

Conceptual design ($\mathcal{CD}$) in this stage, the design is mostly a matter of data modeling, so variability (variation points) concerns (Fig.2) the formalism of the data model, and the model itself [8,15] whereby the elements can be divided into two categories: (i) the core ones which must be present whatever the application is, and (ii) the variable ones, which depends on the application/client.
Framework instantiation: $\mathcal{FM}_{\mathcal{CD}} =< \mathcal{F}_{\mathcal{CD}}, \mathcal{IC}_{\mathcal{CD}} > | \mathcal{IC}$ will be tackled further below.$\mathcal{F}_{\mathcal{CD}} =<mandatory, \mathcal{FG}_{and}, \varnothing > |\mathcal{FG}_{and} = \{f_{Formalism}, f_{Schema}\}|$ $f_{Schema} =<optional, \varnothing, schema>, f_{Formalism} =<mandatory, \mathcal{FG}_{or}, schema> |$ $\mathcal{FG}_{or} = f_{er} \vee f_{uml} \vee f_{express} \vee f_{semantic}|$. $f_{er} =<undefined, \varnothing, \varnothing >$, then f_{er} is a variant. t is undefined because f_{er} belongs to $\mathcal{FG}_{or}$. Note that *schema* is the input of both features *Formalism* and *Schema*, as explained in §. 5.1
Logical design ($\mathcal{LD}$) the variability inside the logical design can emerge mainly at two different levels: data model and normalization (Fig. 2). Mapping rules variants is merged with the data model for the sake of clarity.
Framework instantiation: $\mathcal{FM}_{\mathcal{LD}} =< \mathcal{F}_{\mathcal{LD}}, \mathcal{IC}_{\mathcal{LD}} > |$
$\mathcal{F}_{\mathcal{LD}} =<undefined, \mathcal{FG}_{xor}, cm > | cm$ is the conceptual model resulting from the previous phase. $\mathcal{FG}_{xor} = \{$*hierarchical, ..., object-relational*$\}$.
Physical design ($\mathcal{PD}$) variability can emerge at different levels (Fig. 3).

3.3 Constraining variability

Two types of dependencies are considered in our framework: (i) strict constraints expressed through the couple of relations: *require/exclude*, and (ii) flexible constraints that can be defined as helpful suggestions expressed through the couple

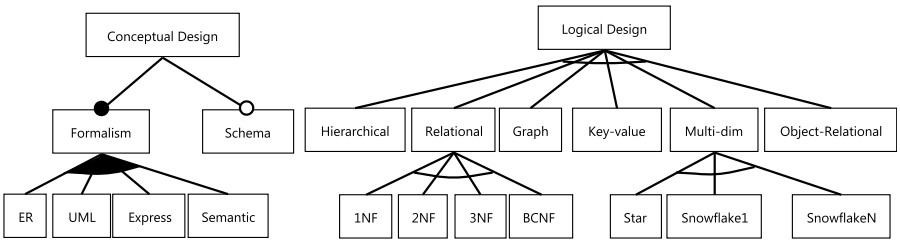

Fig. 2. Excerpts of $\mathcal{CD}$ (Left) and $\mathcal{LD}$ (Right) feature models.

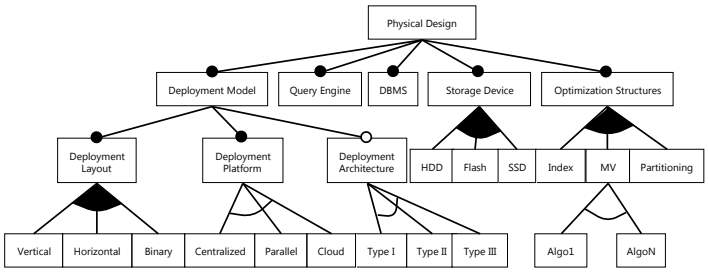

Fig. 3. Excerpt of the $\mathcal{PD}$ feature model.

recommend/recommend-not, and do intervene in order to advise and assist the designer during the configuration process. They are collected from designers' experience, and they can be enriched over time thanks to the designers collaboration. Dependencies are an important element for the configuration process held by users, since they reduce inconsistency and possible configurations number.
Framework instantiation: below, some few examples of our SPL integrity constraints. $\mathcal{IC}_{\mathcal{CD}}$ =<*semantic* ***require*** *deployment-architecture*>
$\mathcal{IC}_{\mathcal{LD}}$ =<*multidimensional* ***recommend*** *vertical*>
$\mathcal{IC}_{\mathcal{PD}}$ =<*flash* ***recommend-not*** *MV*>, $\mathcal{IC}_{\mathcal{PD}}$ =<*MySQL-DBMS* ***exclude*** *MV*>

3.4 Implementation of $\mathcal{DB}$ SPL Framework

We have chosen the *FeatureIDE* tool, an eclipse-based IDE, and one of the most complete and open source plugins that we have found in literature[5]. We have implemented our framework using the *AHEAD* composer that supports composition of *Jak* files. *Jak* extends Java with keywords for FOP.

As depicted in Fig.4, first, features and constraints are modeled via the editor. *FeatureIDE* creates two directories: *features* that contains a directory for each defined feature, and *configs* that can contain different configurations (valid combination of features). For instance, if the user wants to implement the *UML* feature, aiming at transforming the ER model into an UML diagram, he must implement the corresponding algorithm (Algo.1) in the *UML* directory. Note that our features can be one of three types: (i) ordinary ones that contain binary code, (ii) interactive that require an input from the user ($\mathcal{I} \neq \varnothing$), and (iii)

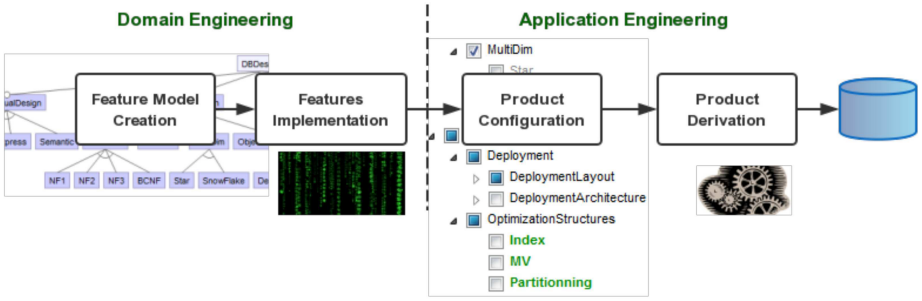

Fig. 4. The development process for software product lines

neutral that contain only parameters used either for evaluating the design or further deploying data.

Once all features are implemented, the user can configure a desired product ($\mathcal{DB}$ instance), by selecting a valid combination of features. While checking some boxes (features), other following boxes can either be: (i) highlighted as a symbol of hints, i.e. recommended choices related to the present feature, (ii) disabled in case of *exclude* dependency, or (iii) automatically checked in case of *require* dependency. We have implemented the basic features and dependencies, that should be enriched by the users by dint of much using.

Finally, the user executes the configuration to derive the final script. This latter corresponds to the functional requirements of the current application. The DBA can enrich it thereafter, so that non functional requirements could be also taken into consideration (stored procedures, triggers, etc.) before the real deployment. This script can be accompanied with additional useful informations as it will be discussed in the next section. SPL input will be defined in §. 5.1.

Algorithm 1: Translation ER models to UML diagram

```
Input: ERM: ER model
Output: UMLD: UML diagram
begin
    foreach entity e in ERM do
        create class c in UMLD;
        foreach attribute a in e do
            create an attribute a' in c;
        end
        foreach instance i of e do
            create an object o of c;
        end
    end
    foreach relationship r in ERM do
        create proper generalization, association, aggregation or unidirectional
        association based on the r relationship types;
    end
end
```

4 $\mathcal{DB}$ Design Evaluation

The final DB design is the result of the different selected features summarized in the script which can also be seen as the path devised by the designer throughout the $\mathcal{DB}$ design feature models. Indeed, it would be much more interesting if the final script is valued in order to help the user to choose between different designs (scripts). Evaluating design process requires two parameters: the criteria of evaluation, and the evaluation tool which can be metrics or cost models to calculate the value. Generally, the criteria used in $\mathcal{DB}$ are performance, energy consumption, and the required size to the implementation.

In our proposal, we're interested in the performance criterion, for which, cost models specifications become as follows:

Cost-model Input: - Queries.
- Data statistics (table sizes, attribute domains, etc.).
- Storage characteristics (Page size, memory size, etc.).
Output: estimation of the number of inputs/outputs between disk and main memory while executing each query.

A cost model is a tool designed to evaluate the performance of a solution without having to deploy it on a DBMS, and to compare different solutions. We recall that we are in the design phase, hence the inputs of the cost model are not set up yet, but designers can always have an idea about the load of frequent queries (considering Pareto principle), and data statistics while analyzing users' requirements. That said, our feature model was conceived with in mind, performance evaluation, that's why we have ignored other aspects as privacy and recovery.

$\mathcal{DB}$ Design Performance Estimation

In practice, the cost model is independent of the used query language. It rather depends on the execution plan of the query, i.e. the order of the algebraic operators in the query tree. These operators are determined by the selected *Logical model*, and each one has its own algebraic formula to calculate its cost. This latter formula, in turn, differs according to the used optimization structures. In a nutshell, each selected feature has an impact on the calculus of the cost model, as follows (in chronological order of the $\mathcal{DB}$ design life-cycle):

- *Conceptual formalism* has no direct measurable impact on performance, since it is requirement-oriented, and a modeling matter more than anything else. It will be of great value if the evaluation criteria was the *security* or *understandability* (*usability*). The former defines which subjects (users, groups, roles) can or cannot access each object. The latter involves metrics for measuring the quality of a conceptual schema from the point of view of its understandability[7].

- *Conceptual schema* determines which query to be considered from the whole workload, since it provides the "local" schema related to current requirements.

- The *Logical Modeling* features determine the algebraic operators to be used.

- *Deployment Layout* features serve to provide a final rewriting of the queries. In fact, queries have to be rewritten according to the final deployment of data.

- *Deployment Paradigm*, *Deployment Architecture*, and *Storage Platform* feed the cost model with necessary parameters, such as storage information: memory size, block size, and number of nodes in parallel environments.

- *Optimization Structures* determine the formula for each algebraic operator.

Allowing designers to value their design choices summarized in the script, implies adding the layer of the above roles in the corresponding features. Algorithm 2 shows the process of valuing the performance of $\mathcal{DB}$ design process.

Now that tools are ready, the best solution is to automatically provide the optimal script value, by choosing from all possible combinations. However, this is a very long and complex task. For the first version of our proposal, we provide a manual solution, that relies on user experience and whereby he has to select the most appropriate paths to his application, and then compare their costs.

Algorithm 2: Calculation of the $\mathcal{DB}$ design performance cost.

Input: - The query workload $\mathcal{Q} = \{Q_1, Q_2, ..., Q_m\}$ (*Requirement Analysis*);
- The set of the selected features (*Designer*);
- $\mathcal{DB}$ statistics (*Requirement Analysis*);
- Storage characteristics (*Neutral* features)

Output: Design cost.

begin

Select the appropriate queries (*Conceptual schema*);
Generate the query trees from the workload (*Logical Model*);
Adjust (rewrite) query trees according to the deployment (*Deployment layout*) ;
Apply the cost model on the final execution plan (*Optimization Structures*);

end

5 Use Case & Experiments

Below, we explain how to use our SPL, and how helpful is the evaluation process.

5.1 $\mathcal{DB}$ Design As an SPL

Fig.5 gives a high-level view of the life-cycle of activities involved in our SPL operation ($\mathcal{DB}$ design). Our starting point is the ER *Conceptual Model*($\mathcal{CM}$), because this formalism can be mapped to any logical model (with user intervention when mapping to *NoSql* models). The user can either create it or provide it. The other formalisms, whenever selected, allow mapping the ER CM to the target formalism. If *Schema* selected, then the user can tailor an existing global conceptual model to specific needs. Note that normalization features are interactive, since they need an extra-input [3] about some design parameters like attribute size, number of tuples, functional dependencies for the different normal forms of the relational model, in addition to hierarchies for the multidimensional case. The designer has definitely at least an approximate idea about these values determined from requirements. Likewise, *Optimization Structures* are interactive features, because they depend heavily on additional parameters like the workload. In practice, some indexes can be derived right from the logical model,

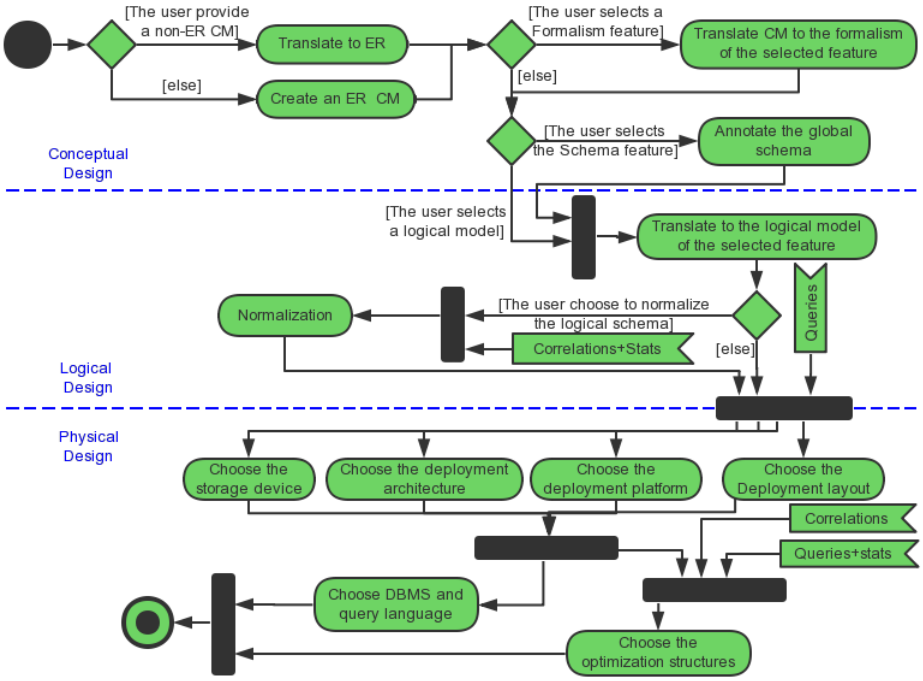

Fig. 5. Activity diagram of the usage of $\mathcal{DB}$ Design SPL

partitioning depends on the query workload, Materialized views depend on both queries and correlations [9]. As for *Deployment Platform*, *Storage Device*, and *Deployment Architecture* features, they are neutral since they only allow to set some parameters without application layer.

After doing these choices, The set of the proposed DBMS shrinks according to the selected features. They can be standard DBMS such as *Oracle*, *PostgreSQL*, or *MySQL*. As they could be, in high-constrained applications, core features of tailor-made DBMS such as those of *FAME-DBMS* for embedded systems [12].

5.2 $\mathcal{DB}$ Design Cost Estimation

In order to show the importance of evaluating the performance of the design process, we instantiate the approach described above using as input the Star Schema Benchmark ($\mathcal{SSB}$). The user has made 6 configurations (scripts). They concern a multidimensional application with different (i) normalization forms: star and a snowflake variant (Fig. 6), and (ii) optimization structures: none, materialized view, and horizontal partitioning. As already mentioned, conceptual design has no direct measurable impact on performance, and we will take the overall schema. Figure7 shows the total performance cost of executing the SSB workload using the six configurations. The designer would choose the one having the lowest performance cost, in this case, it is the one corresponding to a materialized snowflake schema.

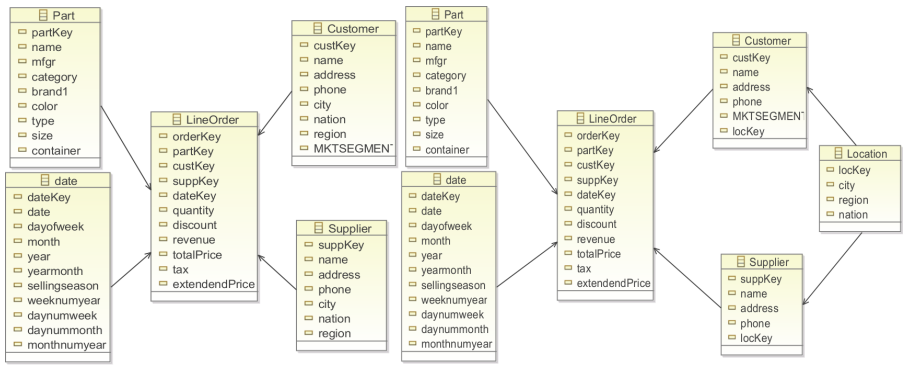

Fig. 6. The $\mathcal{SSB}$ schemes used for experiments.

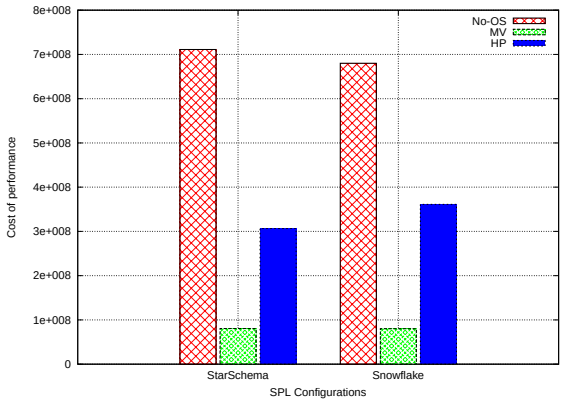

Fig. 7. The performance cost of selected scenarios.

6 State of The Art

Originally, $\mathcal{DB}$ systems were relatively static, uniform and any required change would demand complex and onerous editing of the source code (e.g. hierarchical DBMS). Over time and giving the upward diversity in $\mathcal{DB}$ application, developers strive for general-purpose solutions. *Oracle* is a famous example of a heavyweight DBMS. Unfortunately, generality often introduces functional overhead to the detriment of performance and clarity and it is inappropriate for use in high constrained environments (e.g. embedded systems). As a result specific-purpose solutions have appeared. They are based on the principle that an application achieves its best performance when it uses a "lean and mean" framework that exactly fits to the special application scenarios. *PicoDBMS* [10], a lightweight DBMS for smart-cards, and *TinySQL*, a tailor-made language for querying sensor networks, best illustrate this type. Specialized solutions often redeveloped huge parts of systems from scratch leading to duplicated implementation efforts,

development costs/time to market, and poor quality of software. Between the two extremes of piling-up a broad range of functionalities and reinventing the wheel for each scenario, both of the above approaches present limited capabilities for managing variability as they do not provide a general approach for reusing functionality to similar systems of a domain. Customizable solutions can be used to provide an extensible architectures or to generate tailor-made $\mathcal{DB}$ that attain high customizability and reuse. They can be built with a number of different techniques [11] as summarized in Table 1. Most of them correspond to the physical layer. Presumably, this is motivated by complexity and diversity present in that phase and performance requirements of $\mathcal{DB}$ applications. DBMS is the most studied part in the physical phase and the conceptual scheme in the others. In the $\mathcal{DB}$ design area, Broneske et al. [4] have shown the superiority of the SPL in mastering variability compared with other design methodologies. Our approach is unique since it tackles the whole design process in one stroke so as to consider the interdependence of the phases whatever the environment is (ordinary as well as high-constrained environments). At first glance, our tool can be assimilated to $\mathcal{DB}$ modeling tools like *PowerAMC*[14], since this latter can also generate a script corresponding to some design choices. However, *PowerAMC* deals solely with the ***modeling aspect*** of the $\mathcal{DB}$ design, while omitting the other design matters as well as their dependencies like the normalization of the logical schema, the physical optimization and so forth. Moreover, *PowerAMC* can only handle relational environments (ROLAP/ROLTP).

Phase	Target	Input	Technique
Conceptual Design	Conceptual Schema	Global Schema	Modeling Tools[14] SPL[8,15]
Logical Design	Query Language	Requirements of a specific logical schema	SPL [13]
Physical Design	Query Optimizer	Requirements of a specific physical schema	SPL [16]
	Deployment Layout	Requirements of a specific data layout	Grammar [18]
	DBMS	Requirements of a specific DB type	Component-based[6],FOP[1] Preprocessors,AOP[17],SPL[12]
Whole $\mathcal{DBLC}$	$\mathcal{DB}$ Design	Requirements of the $\mathcal{DB}$ application	SPL

Table 1. Classification of customization solutions to design $\mathcal{DB}$.

7 Conclusion

In this paper, we've been interested in managing variability of $\mathcal{DB}$ design, the most critical step of $\mathcal{DB}$ Development Life-Cycle (analysis, design, implemen-

tation, testing, and tuning). This was motivated by the large functionality and complexity of today's $\mathcal{DB}$ applications.

Studying variability in the design phase affect the whole $\mathcal{DBLC}$, and its scope can be easily extended to cover the remainder. In fact, our second contribution falls under the testing phase. Also, we are further going to address the evolution issue responsible on the *tuning* phase, that can be well handled thanks to the development overview provided by the variability study of the design step.

References

1. Batory, D., Barnett, J., Garza, J., Smith, K., Tsukuda, K., Twichell, B., Wise: Genesis: an extensible db management system. Software Engineering, IEEE (1988)
2. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach. ACM Press/Addison-Wesley Publishing Co. (2000)
3. Bouarar, S., Bellatreche, L., Jean, S., Baron, M.: Do rule-based approaches still make sense in logical data warehouse design? In: Advances in Databases and Information Systems - 18th East European Conference, ADBIS (2014)
4. Broneske, D., Dorok, S., Köppen, V., Meister, A.: Software design approaches for mastering variability in database systems. In: GvDB. vol. 1313 (2014)
5. Dammagh, M.E., Troyer, O.D.: Feature modeling tools: Evaluation and lessons learned. In: ER Workshops. pp. 120–129. LNCS, Springer (2011)
6. Geppert, A., Scherrer, S., Dittrich, K.R.: Kids: Construction of database management systems based on reuse. Tech. rep. (1997)
7. Golfarelli, M., Rizzi, S.: Data warehouse testing: A prototype-based methodology. Information & Software Technology 53(11), 1183 – 1198 (2011)
8. Khedri, N., Khosravi, R.: Handling database schema variability in software product lines. In: 20th Asia-Pacific Software Engineering Conference, APSEC (2013)
9. Kimura, H., Huo, G., Rasin, A., Madden, S., Zdonik, S.: Coradd: Correlation aware database designer for materialized views and indexes. PVLDB 3, 1103–1113 (2010)
10. Pucheral, P., Bouganim, L., Valduriez, P., Bobineau, C.: Picodbms: Scaling down database techniques for the smartcard. The VLDB Journal 10(2-3) (Sep 2001)
11. Rosenmller, M., Apel, S., Leich, T., Saake, G.: Tailor-made data management for embedded systems: A case study on berkeley db. DKE 68(12), 1493 – 1512 (2009)
12. Rosenmüller, M., Siegmund, N., Schirmeier, H., Sincero, J., Apel, Leich, Spinczyk, Saake, G.: Fame-dbms: Tailor-made data management solutions for embedded systems. In: Software Engineering for Tailor-made Data Management (2008)
13. Rosenmüller, M., Kästner, C., Siegmund, N., Sunkle, S., Apel, S., Leich, T., Saake, G.: SQL à la Carte – Toward Tailor-made Data Management. In: 13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW). pp. 117–136 (2009)
14. SAP, S.: Sap sybase powerdesigner. `http://www.sybase.com/products/modelingdevelopment/powerdesigner` (2013)
15. Siegmund, N., Kästner, C., Rosenmüller, M., Heidenreich, F., Apel, Saake: Bridging the gap between variability in client application and db schema. In: 13. GI-Fachtagung Datenbanksysteme für Business, Technologie Web (BTW) (2009)
16. Soffner, M., Siegmund, N., Rosenmüller, M., Siegmund, J., Leich, T., Saake, G.: A variability model for query optimizers. In: DB&IS. pp. 15–28 (2012)
17. Tesanovic, A., Sheng, K., Hansson, J.: Application-tailored database systems: a case of aspects in an embedded database. In: Database Engineering and Applications Symposium, 2004. IDEAS '04. Proceedings. International (2004)
18. Voigt, H., Hanisch, A., Lehner, W.: Flexs a logical model for physical data layout 312, 85–95 (2014), `http://dx.doi.org/10.1007/978-3-319-10518-5_7`