Towards a model-based collaborative framework for calibrating database cost models

Abdelkader Ouared[1,2], Yassine Ouhammou[3], and Ladjel Bellatreche[3]

[1] Ibn khaldoun University of Tiaret, Algeria
[2] National High School for Computer Science (ESI), Algiers, Algeria
[3] LIAS/ISAE-ENSMA, Futuroscope, France
a_ouared@esi.dz
{yassine.ouhammou, bellatreche}@ensma.fr

Abstract. The database systems optimization is often driven by using cost models during the physical-design phase. A large panoply of database cost models have been proposed, where each one depends on various kinds of systems parameters (platform, software system, database schema, etc.) and can be related to various resources (e.g. processors, memories, networks). Currently, the development of cost models is still time-expensive due to the time allowed for calibrating and tuning parameters in order to obtain cost models close to the practical usages of database systems. Moreover, a cost model can concern simultaneously many resources, hence its development may require to be done in a collaborative way between actors who complement each-other since each one of them can be expert in a particular resource domain.
In this paper, we propose a framework which helps cost model designers to work collaboratively and enables them to automatize and ease the traditional work-flow dedicated to calibrate cost model parameters through open-source DBMS (database management systems). This framework is compliant with PostgreSQL and it is implemented as model-based infrastructure thanks to model-driven engineering settings.

1 Introduction

Evaluating the performance of database systems has become more and more primordial in the era of big data. It helps to check if the non-functional requirements of a database system are met or not. This evaluation can be performed in two ways. The first one is the application of real experiments on the system under design. The second one consists on simulating experiments by using formal mathematical models enabling the quantification of metrics (e.g., response-time, energy, throughput). These models are called cost models. In this paper, we focus on the work-flow of the cost models development.

1.1 Context and work positioning

Nowadays the development of database systems deals more and more, in addition to functional requirements, with non-functional requirements (like response delays and energy consumptions of queries). In order to obtain database systems that meet these

In: C. Cabanillas, S. España, S. Farshidi (eds.):
Proceedings of the ER Forum 2017 and the ER 2017 Demo track,
Valencia, Spain, November 6th-9th, 2017,
published at http://ceur-ws.org

kinds of requirements, several physical optimization structures (e.g., indexes, partitions, materialized views, etc. [2]) can be used depending on the DBMS (database management systems), which is selected for the database system and also on the database architecture [1]. However, the selection of the appropriate physical optimization structures is np-hard since it is considered as a decision problem [2]. Thus, to reduce this complexity several researches have adapted the optimization algorithms [7] [2] (e.g., greedy algorithms, evolutionary algorithms, linear programming algorithms) in order to find optimal solutions (i.e. physical optimization structures). These solution can not be approved by database system-designers without using appropriate cost models that quantify the benefit of the selected physical structure.

Each cost model integrates the maximum of information to provide a precise measure. This information is usually related to the system configuration: the database (e.g. size of tables, number of tables, etc.), the workload (e.g. number of joins, selectivity factors of predicates), the deployment platforms (e.g. centralized, parallel, cloud) and the storage layout (e.g. column store, row store). Since 1980s many cost models have been proposed, each one is dedicated to a specific database system configuration, and allows to calculate a specific metric (e.g., response-time, energy, throughput). Besides, the technology evolution motivates researchers to adapt and revisit existing cost models by integrating new configuration parameters brought by that evolution. An interesting manner that enables researchers to get adapted cost models is when this adaptation is backed by the usage of an open-source DBMS, such as postgreSQL[4]. Indeed, cost model developers adapt the DBMS source-codes of existing cost models in order to well calibrate parameters, hence obtaining new refined cost models. Figure 1 sketches the work-flow of the cost model construction based on the usage of an open-source DBMS.

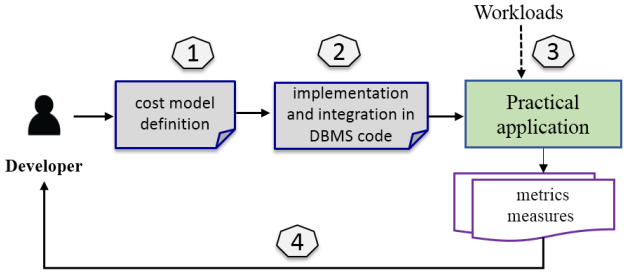

Fig. 1. Conceptual work-flow of the cost model development

First, the cost model developer(a.k.a. modeller or designer) defines a blueprint version of a given cost-model (Step 1 of Figure 1). Then, he/she implements the cost model as a program-code based on the DBMS code (Step 2 of Figure 1). Hence, this new implementation can be launched in order to get measures of desired metrics depending on a given workload (Step 3 of Figure 1). Based on these measures the developer can calibrate the former cost model and then defines a new refined one (Step 4 of Figure 1). In this step, the cost model designers can be assisted by various techniques, such as machine learning settings, to tune parameters and also to deduce their inter-dependencies. All these steps can be repeated many times till obtaining a cost model that matches the developers needs.

[4] PostgreSQL is a relational database system with a fairly traditional architecture.

1.2 Problem statement

As it is shown in the process illustrated in Figure 1, the development, calibrating and adaptation of cost models can be laborious and time-consuming since the process is not automatic (i.e. it is done manually) and it requires to be repeated many times. Moreover, every iteration of this classical process can be error-prone, due to complexity of cost models parameters (algorithms, platform characteristics, database features) which have a significant impact on the quality and the correctness of the cost models[14].

This current manner of development is also penalizing the collaboration between designers of the same cost model when the development of this latter requires different domain experiences. On the one hand, the collaboration is hard to be done synchronously. That is, merging, managing conflict and versioning actions have to be done manually, which can be error-prone. On the other hand, there is no standard language enabling to have a cost model as an abstract description in a machinable format.

1.3 Paper contributions

There has been a lot of work vastly explored building and adapting database cost models. However, to the best of our knowledge, there is no work that provides an infrastructure helping designers to define and to adapt cost models collaboratively.

To fill this gap, in this paper we propose a framework that allows cost model designers to define and to generate automatically the appropriate programs based on source-code of PostgreSQL. The framework offers three main services: (i) the description of cost models, (ii) the multi-utilization to ensure synchronous collaboration, and (iii) automatic code generation. Our proposition is based on model-driven engineering settings. Hence, we take benefit from our recent work called CostDL (cost model description language [19]) which is a description language dedicated to express cost models as formal abstractions. Also, we use model repository called CDO (Connected Data Objects) [10] to ease the management of conflicts related to the collaboration and versioning. The automatic code generation is based on a transformation process which enables to transform cost models to code that corresponds an open-source DBMS, which is PostgreSQL in this paper.

1.4 Paper outline

The rest of the paper is structured as follows. Section 2 presents the background. In Section 3 we present our proposed approach. Section 4 highlights the implementation of our proposition and shows its applicability and its benefit. Section 5 is dedicated to discuss the related work. Finally, Section 6 summarizes and concludes this paper.

2 Background

This section presents the scientific baseline of this paper. First, we give a brief description of cost models advances. We set out challenges faced by cost model designers through a discussion. Then, we introduce the benefits of the model driven engineering paradigm. Moreover, a description related to our recent work [19] is provided and its interest is motivated and illustrated by an example.

2.1 What is a cost model?

Since 1980s, a lot of cost models ($\mathcal{CM}$s) have been proposed (e.g. [22, 16, 23, 4]). We summarize some of their principles hereafter.

Regularly, database end-users manipulate different kinds of operations (such as: *join*, *scan*, *sort*, *etc.*), where each operation execution costs in terms of response-time, memory-size and/or energy. The cost of an operation is related to the computational complexity, the platform characteristics and various database features. Thus, each $\mathcal{CM}$ is dedicated to be used in a particular context characterized by parameters related to the architecture layers (for instance: operation system layer, data system layer, access method layer, buffer manager layer and storage system layer) [5]. Moreover, a $\mathcal{CM}$ is also characterized by a mathematical formula, which depends on parameters presented in the context (such as: database size, indexes, disk-layout). This mathematical formula may be derived from a set of other basic math formulas, where every one represents a logical or physical cost. While the former is related to properties of data processing regardless of the deployment layout, the latter quantifies the impact of the hardware parameters such as memory-size and block-size [16].

2.2 Discussion and motivations

The underlying idea behind this paper was inspired by our experience in developing cost models. In our laboratory, we use to develop cost models where the majority of them are developed from scratch. For instance, we recently proposed cost models in [20] [21] by using the query optimizer of the PostgreSQL. The development of such cost model lasts for several weeks for various reasons. (1) We have studied the C-programs of PostgreSQL before coding manually the cost model. (2) For a refinement reason, we have repeated this coding process several times until achieving a good execution performance and a good result quality. (3) While the design of the cost model, in question, concerned different aspects (e.g., database and hardware), we worked together in order to help each-other and to combine our contributions. However, because of the hard-coded approach that we followed to develop our cost models, we were obliged to progress sequentially and alternatively instead of progressing in parallel and simultaneously. (4) Besides, even if some solutions of code versioning like Git [6] and SVN [8] ease to manage the conflicts in case of multi-developers, its utilization requires to be familiar with the code produced by other collaborators which is not intuitive. Unfortunately, these enumerate reasons are the common difficulties and challenges that cost model designers face. Moreover, that also penalizes the reuse of these efforts. For instance, if one decides to replace PostgreSQL by another open-source DBMS, one has to code again a program according to the programming language and the source-code of this DBMS.

2.3 Model-driven engineering

MDE (Model-driven engineering) [13] is a paradigm that makes the systems software development focusing on the utilization of models as a cornerstone. Hence, models are considered as centric-entities of a generative process. MDE allows the overcoming of the growing complexity of systems. In other words, MDE offers the possibility to describe a system with the help of a series of models. More abstract models are refined into more concrete models by model transformations that ensure that the information from higher-level models is retained in lower-level models [11].

2.4 Existing work: CostDL in a nutshell

In order to define $\mathcal{CM}$s in a structured way and also to be able to compare them, every $\mathcal{CM}$ must be a software entity. Consequently, we have used MDE settings and have proposed *CostDL* which is a cost models description language [19]. *CostDL* is a meta-model (see Figure 2) that allows $\mathcal{CM}$ designers to define their models based on several parameters related to the database, queries, and the platform. *CostDL* includes MathML [3] in order to help designers to define formally all mathematical functions of a $\mathcal{CM}$. A tool has been developed and it allows designers to express $\mathcal{CM}$s, where each one is an instance conforms to the *CostDL* language.

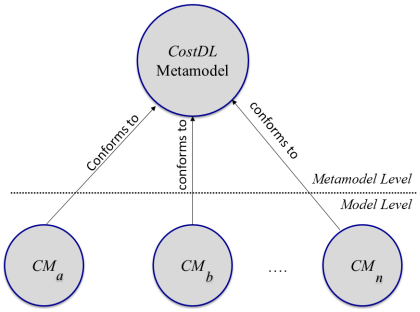

Fig. 2. Meta-modeling and modeling levels

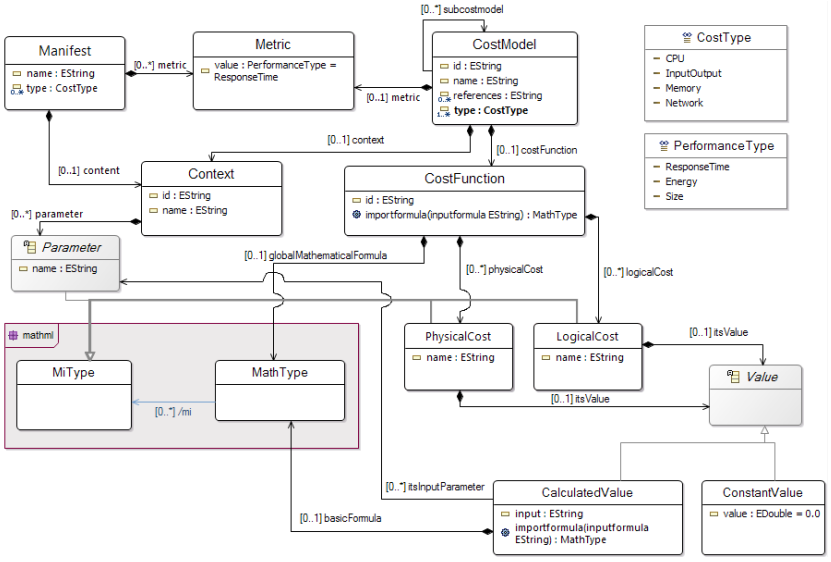

Fig. 3. Excerpt of CostDL meta-model: core entities

Example In the following, we present a cost model $\mathcal{CM}_{example}$ that has been proposed by D. Bausch et al. [4]. Let $Cost^{I/O}(hashjoin)$ be the cost formula of $\mathcal{CM}_{example}$. It

enables to measure the I/O response-time for hash-join operations, where:

$$Cost^{I/O}(hashjoin) = \| R_i \|_p c_{rw} + \| R_i \|_p c_{sr} + \| R_o \|_p (c_{rw} + c_{sr}) \quad (1)$$

Figure 4 shows the expression of $\mathcal{CM}_{example}$ in *CostDL* language. The instance, that conforms to the *CostDL* meta-model, contains the different parameters related to the $\mathcal{CM}_{example}$ context. Figure 4 also shows that the cost function is composed by a set of logical and physical costs which are inputs of the math formula.

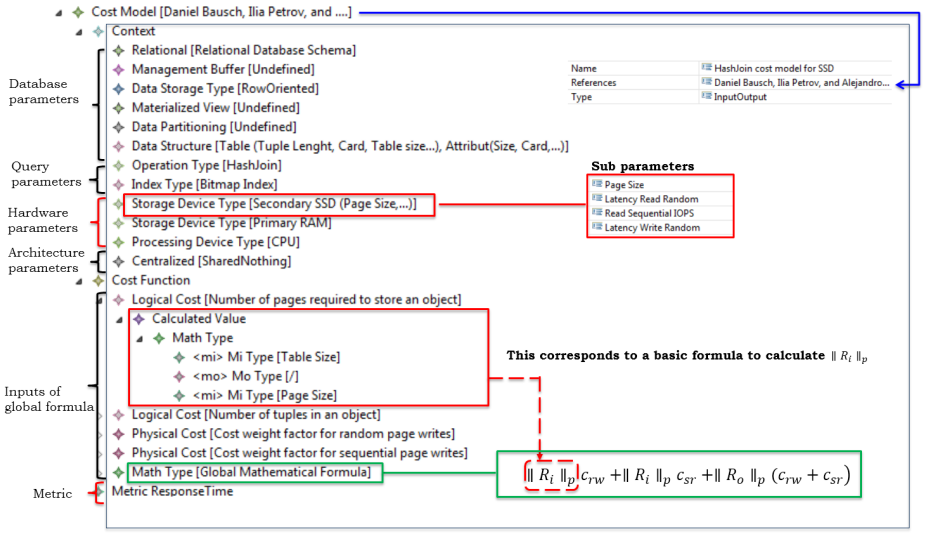

Fig. 4. Excerpt of the cost model $\mathcal{CM}_{example}$ expressed in CostDL

This section summarized the background helping to ease the understanding and leading to support the contributions. Thus, the contributions of this paper aim to capitalize efforts provided for designing cost models. First, we aim to automatize the process by using MDE techniques to take benefits from the model abstraction, the code generation and the reuse capabilities. Secondly, we aim to ease the management of collaboration conflicts by rising them from the codes up to modeling level.

3 The Conceptual Framework for calibrating CMs

This section is devoted to present the contribution materials. First of all, we will present an overview of the framework called *CF-CM* (Conceptual Framework for Calibrating cost Models). Then we present our approach to examine and synthesize concepts existing in to PostgreSQL programs. In addition, we will expose the mapping relations in order to bridge the gap between cost models expressed in CostDL language and their C-codes in PostgreSQL. Finally, we will explore different solutions that can help us the manage the collaboration before opting for the appropriate one.

3.1 *CF-CM* Overview

Since the paper contributions have led to the development of the framework called *CF-CM*, we first give an overview of the capabilities that it provides. *CF-CM* offers

the following services: (i) Service 1: edition of cost models expressed in a machinable formalism. (ii)Service 2: generating programs from any cost model expressed by Service 1. The generated programs have to be compliant with open-source DBMS. (iii) Service 3: in case of a collaboration, this service helps users to combine their modifications to be included in a global cost model.

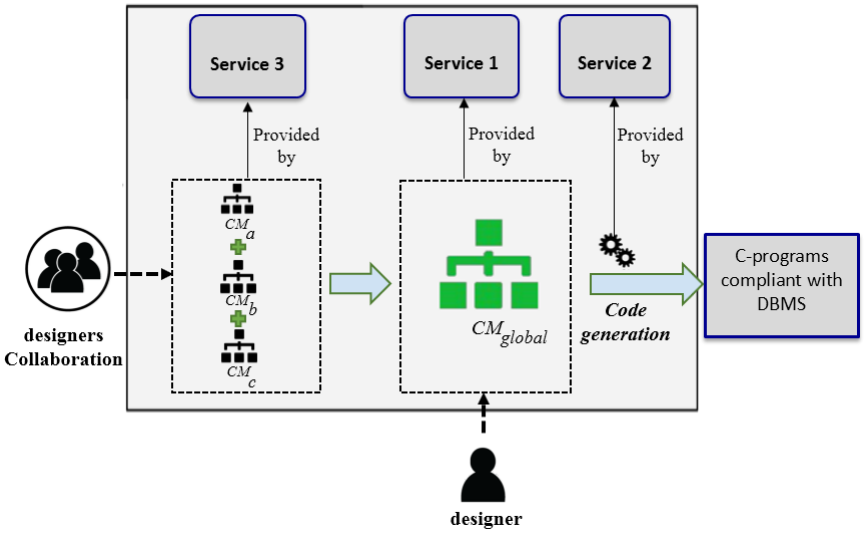

Fig. 5. Overview of the *CF-CM* Infrastructure

Figure 5 shows all these services and how they communicate to each other. Indeed, Service 1 is based on the *CostDL* language [19]. Then, this service allows expressing the cost models as instances of the [19] meta-model. That will help to reuse, compare and share the cost models under the same formalism by downloading and uploading them. Service 1 is dedicated to be used by only one designer, but we can have many instances of Service 1 as collaborator-designers. In this case Service 3 is required. That is, every designer can download the last version of the cost model. Then, the designer can modify it and commits the updated version at run-time. Service 3 guarantees to impact all cost model instances edited by the distributed collaborators. The advantage of this service is to manage the conflicts at a model-level instead of comparing codes at a program-level. The benefit is that during the modeling process we assume that all collaborators' mastery and understand *CostDL* language since it contains a limited set of concepts which are common to all designers. Another important point is that contrary to programs where the modifications may impact several code-pieces scattered in different files, the modifications through our framework impact only one file. Once a draft version of a cost model is obtained, designers can transform it to programs. This transformation is provided automatically by Service 2. The transformation is based on two meta-models. The first one is the *CostDL* meta-model which represents the generation source. The second one is the meta-model of the PostgreSQL API (Application Programming Interface) which is the generation target (see Section 3.2). The transformation does not generate only programs, but it also integrates them by modifying the appropriate PostgreSQL files according to the input cost model. While Service 1 is based on our recent existing work, in the following we will focus on the cornerstone elements of Service 2 and Service 3.

3.2 CostDL to PostgreSQL transformation

In this section we present the *CostDL* to PostgreSQL transformation, which is the second step in the approach presented in Figure 5. We first introduce the PostgreSQL API metamodel and then, we focus on the transformation itself.

PostgreSQL API In order to transform any cost model conforms to *CostDL* language to PostgreSQL *C-programs*, we have first studied the structure the PostgreSQL API (Application Programming Interface). The underlying ideas behind this study is to achieve two objectives. The first one consists in making the transformation as automatic as possible by targeting key API elements. The second objective is to have a flexible transformation that can be easily extended. Therefore, the skeleton of the PostgreSQL API has been summarized as it is shown in the model illustrated in Figure 6.

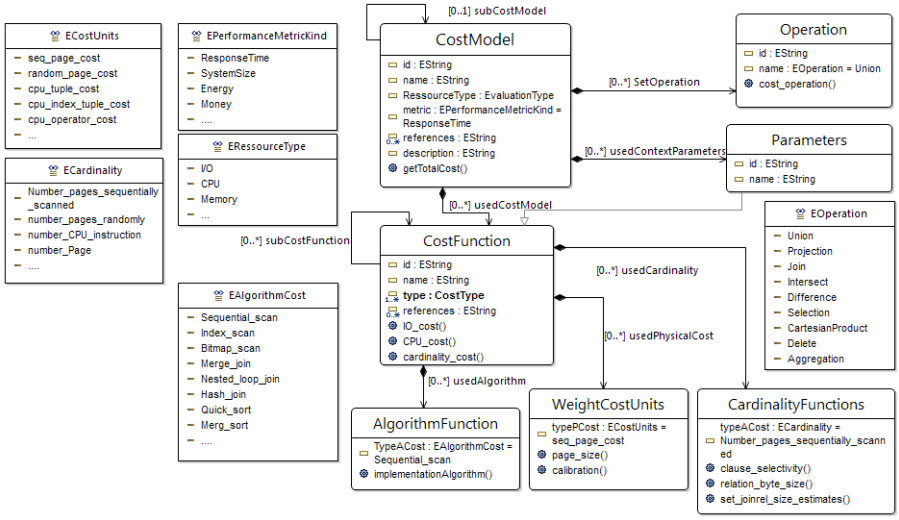

Fig. 6. Excerpt of PostgreSQL API conceptual model

It shows the conceptual model of API skeleton that is composed of a set of classes to manipulate DBMS PostgreSQL. In this metamodel, the global mathematical formula may be derived from a set of other basic math formulas, where everyone represents a logical, physical or algorithm cost. While the cost model to compute a performance metric of a given query by assemblies their costs [16]: (i) **Logical Costs** Estimating the size data volumes of a given database operations, we distinguish three data volumes: input (per operand), output, and temporary data or data stream. Data volumes are usually measured as cardinality, i.e., number of tuples. (ii) **Algorithmic Costs** extend logical costs by taking also the computational complexity (expressed in terms of *O-classes*) of the algorithms into account (NestedLoop-Join, Hash-Join,...). (iii) **Physical Costs** finally combine algorithmic costs with system/hardware (CPU, I/O and Net) specific parameters to predict the total costs in terms of execution time.

The CosDL Language Each user uses common parameters to describe his/here cost models. In our scenario, the initial instance cost model would conform to *CostDL*

language dedicated to describe database cost models [19]. A cost model $\mathcal{CM}$ is characterized by four elements: its **cost types**, its **context**, its **cost function** and its performance **metric** kind.

A class diagram cost model is defined as a tuple $\mathcal{CM}_i =< CT_i, cxt_i, func_i, m_i >$, where $\mathcal{CM}$ is the set of classes and is characterized by four elements: its cost types, its context, its cost function and its performance metric kind. $\mathcal{CM}$ calculate the value of the cost model metric $m_i \in$ {Response time, Energy, Size}. The cost type CT_i represents the database component considered by the cost model, then the $CT_i \subseteq \{CPU, I/O, Memory, Network\}$ set.

A context is a set of parameters $cxt = \{p_1, p_2, .., p_n\}$. Each $p_i \in P = P_{Database} \cup P_{Hardware} \cup P_{Architecture} \cup P_{Query} = \bigcup cxt_i$, where every P_j is a set of parameters that belong to one of the four categories. Moreover, $P_{Database} \times P_{Hardware} \times P_{Architecture} \times P_{Query} = \{cxt_1, .., cxt_m\} = CXT$ is a set of all possible contexts of cost model universe.

The cost function $func_i$, of a given cost model $\mathcal{CM}_i$, permits computing the value of the cost model metric m_i based on a subset of the context parameters cxt_i. The cost function is a tuple $func_i =< Param_i, mf_i >$. $Param_i \subseteq cxt_i$ (subset of the cost model context), and mf_i is the mathematical formula, where $mf_i : P^k \rightarrow \mathbb{R}_+$ and $k \leq$ cardinal$(Param_i)$.

***CF-CM* Transformation** The *CF-CM* process is the code artifacts generation that creates a corresponding *C-class* via the Blueprints API (see Fig. 6). By using the Blueprints API, the *CF-CM* generator creates a cost model that contains the parameters and formulas. We provide a list of mapping rules necessary derive cost model from CostDL concepts (see Tab 1).

Table 1. CostDL to A Cost Model of PostgreSQL

CostDL Concepts	**PostgreSQL Concepts**	**Comments**
Algebra Operation Op	relational algebra operations	one-to-one correspondence
Physical Parameters : $p_i \in (P_{Hardware} \cup P_{Arch.})$	cost units: c_s, c_r , c_t , c_i, and c_o	expressing physical parameters of the machine
Logical,Parameters: $p_i \in (P_{Database} \cup P_{Query})$	Selectivity parameters: n_s, n_r , n_t , n_i , and n_i	statistical values of database (e.g. tuples, width)
LogicalCost	void set_joinrel_size_estimates() double relation_byte_size() - Selectivity clause_selectivity() - void set_joinrel_size_estimates() double relation_byte_size() void initial_cost_operation()	$assign_{op,alg}(card, f_{card,op})$ Assign cardinality functions for basic operations
PhysicalCost	IO-Cost() CPU-Cost()	$assign_{op,alg}^{IO,CPU}(Cost, f_{IO,op})$ methods to assign CPU/IO cost to each basic operation depending also on the used implementation algorithms.
=,>,>=,<,<=,<>	=,>,>=,<,<=, !=	one-to-one correspondence
+, -,/, %, *	+, -,/, %, *	
and,or,not	&&,||, !	
variable	variable	
literals	literals	

These mapping rules have been operationalized in the model-to-model transformation. This translation is done using plug-in *Acceleo* available within Eclipse environ-

ment. In this work the instance is then translated into an *C-program* using model to text transformation process. Thanks to the interoperability facilitated by MDE, we can imagine a variety of further usages of cost models. We assume that users can develop user-friendly tools to exploit shared cost models. For instance, in our laboratory we have developed a third party tool that transforms XMI files corresponding to cost models to a *C-program* based on the PostgreSQL API (see Fig 6).

Listing 1.2 shows the generation code of the example presented in fig. 4 (see Section: 2.4). At the end of the design, one can check the conformity of the cost model. For this, a set of structural rules have been injected in the meta-model. These rules are expressed as OCL (Object Constraint Language)[18] invariants. Listing 1.1 is an example of a structural rule. It means that all physical costs and logical costs, which are inputs of a given cost function, have to be referenced as `MiType` instances in the `MathType` (see [19] for more details) instance of that cost function.

```
Class CostFunction
self.globalmathematicalformula.mi->includesAll(self.logicalcost)
and self.globalmathematicalformula.mi->includesAll(self.physicalCost)
```

Listing 1.1. An OCL structural rule

This generator tool has been tested in [20][5]. That is, by using Acceleo language we have developed a code generator which generates *C-programs* based on PostgreSQL API. The objective is to generate the C code of every cost model conforms to *CostDL* [19]. As an example, the $\mathcal{CM}_{example}$ of the running example section (Section 2.1) is translated to functions implemented in C to measure the cost of the hash-join operation. The following listing represents an excerpt of the generated result.

```
    ...
void costhashjoin(Path *path, PlannerInfo *root,
RelOptInfo *baserel, ParamPathInfo *param_info)
{
 Cost   startup_cost = 0;
 Cost   run_cost = 0;
 double   spc_seq_page_cost;
 QualCost qpqual_cost;
 Cost   cpu_per_tuple;
 Assert(baserel->relid > 0);
 Assert(baserel->rtekind == RTE_RELATION);
    ...
 if (param_info)
 path->rows = param_info->ppi_rows;
 else  path->rows = baserel->rows;
 if (!enable_seqscan)
 startup_cost += disable_cost;
     ...
```

Listing 1.2. An excerpt of C generated code

4 Proof of Concept and Tooling

To stress our approach and to proof how it is useful and helpful, this section is devoted to present a global usage scenario of the *CF-CM* framework. In parallel, technical implementations are highlighted. The usage scenario is organized as it is shown in Figure 7.

[5] `http://www.lias-lab.fr/forge/projects/ecoprod`

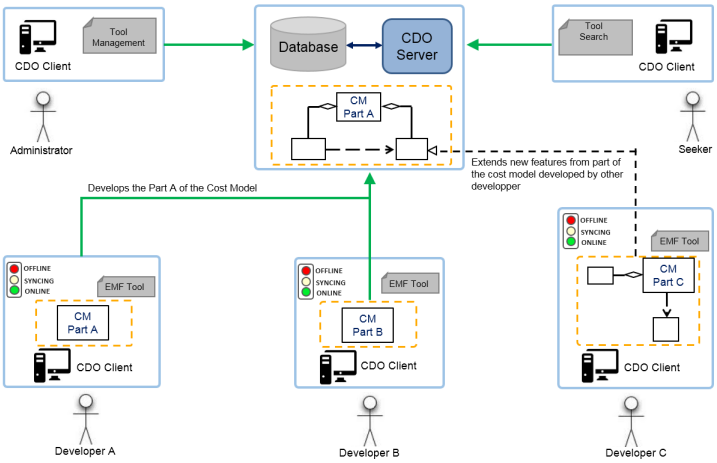

Fig. 7. Technical implementation

4.1 Tool Support of *CF-CM*

Our approach has been implemented and made available as web system based on Eclipse Tooling Support. Eclipse projects have become a standard because it supports the main technologies from OMG. Eclipse tools have with EMF models that support DSL Tools and collaborative work on distributed model repositories like EMFStore[6] and CDO[7]. It supports several DBMS, including relational and NoSQL databases. Our implementation is based on open source project: Connected Data Objects (CDO). This technology provides support to establish remote repositories and the functionalities to work with them ensuring consistency and security is needed. CDO is a semi-automated persistence framework that works natively with *Ecore models* and their instances.

We implement its common functionality by adapting Tooling CDO for: Storing/ sharing cost models, live collaboration with CDO and versioning support and comparison. The four pillars of the proposed infrastructure for *CF-CM* are: (i) Cost Model Editor Service, (ii) Sharing/ Collaboration, (iii) Code Generating and (iv) Repository Management. To ease the understanding, we provide the URL of a demonstration video of the project [8].

Cost Model Editor Service The cost model editor is composed of several user interface. It manages cost models in the repository, thus it permits users to upload, download, delete cost models, and even search cost models conforming to a specific metamodel. The cost model editor service permits users to install them locally, at this stage of the project cost model we do not refer to on-line. By creating a local clone of a repository. This repository is synchronized with the remote repository as soon as the connection is restored. Cost Model Editor Service provides two major services: seeking cost models and sharing cost models. There are two possible ways to create a cost model expressed in CostDL: via an offline editor or via an online Editor. The offline

[6] http://www.eclipse.org/emfstore
[7] http://www.eclipse.org/cdo
[8] The demonstration video of our tool is available at: https://youtu.be/rHIVvJEOrbM

editor is a tool based on Java EMF (Eclipse Modeling Framework) API and has been integrated as a plugin-in Eclipse [9] which is an integrated development environment (IDE). Through the editor tool, every cost model instance is saved as an XMI (XML Metadata Interchange [9]) file.

Collaboration and Sharing CDO is Huge Ecosystem on top of EMF, another advantage of model repositories is the possibility to work collaboratively on cost models. A distinction between offline and online collaboration (on the Web) can be made. Offline collaboration can be regarded as the classical process for source code versioning. A cost model can be shared and checked out afterwards. Then changes can be made and committed. The usage scenario is organized as it is shown in Figure 7. The cost model is fragment into smaller components ease the development because every developer can be specialized in a specific part of components (CPU, I/O, Network, Join, sort etc.) and they make different actions in the cost model (calibration, extended, adapted). Consequentially, user has the ability to load only parts of the cost model that are required.

Repository Administration CDO provides an API [10] to manage repositories remotely. We exploit the API provided by CDO and integrate these functions with a graphical user interface to ease the repository. In order to manage the security system, an entity which will create user accounts, will change passwords and will set user rights is needed. The security manager needs to have a special account to log into the system. In addition, to manage the repositories, this role has to be in charge of to create and delete repositories About cost model versioning, old revisions of cost models can be restored. CDO implements the *Audit View* which can offer a way to get versioning. It is possible with this *CDO View* to see previous versions of the cost models in the repository, it would be possible to use a previous version if it is necessary. Based on CDO revisions, CDO supports the concept of optimistic versioning. This means that An developer first change the state of cost model locally before resynchronising the changes with the server witch keep a history of the different states of the cost models. Note that it's possible to save or export an old state of the model in an XML file and load the model if it is necessary.

5 Related Work

The cost models in database systems have become an active research topic. There has been a plethora of studies and initiatives by the research community. While reviewing the literature of works considering the $\mathcal{CM}$s consumers are usually researchers, industrials and students. Usually, the $\mathcal{CM}$ designed and developed from scratch, we call this design approach the *Hard-coded approach*. Therefore, the $\mathcal{CM}$s must be newly designed, implemented, and tested which leads to duplicate implementation efforts, and thus, increased development costs; so, the possibility of errors is rather high. This scenario, in which a developer or system designer wants to create a $\mathcal{CM}$ from scratch. Specifically, this can be pushed by (i) the advance in database technology (e.g. column store, parallel execution) (ii) the progress in hardware technology (e.g. new CPU or GPU) (iii) or the upcoming of new NFR (e.g. the energy, system sizing). Thus, this is the

[9] Eclipse Modeling Project. www.eclipse.org/modeling/

[10] org.eclipse.emf.cdo.common.admin

most challenging and resource intensive scenario, the developer or the designer have to revisit all layers impacting in the system, and to extract the relevant parameters with their respecting basic units to build accurate $\mathcal{CM}$s, in other hand development process of $\mathcal{CM}$ requires a deep knowledge related to many aspects: databases, hardware, machine learning for calibrating its parameters, statistic to estimate parameters such as selectivity factors of join predicates, etc. For example, we can cite some advanced studies and tools developed in the top of PostgreSQL: the tool *Parinda* [15] that offers designers an interactive physical designer for PostgresSQL, *PET* - an energy-aware query optimization framework [24], the work of [14] that challenges certain theories regarding query optimizers, and *OntoDBench* for evaluating semantic databases [12], and *SQOWL* - a tool to performing type inference over data stored in PostgreSQL [17]. Certainly, we cannot deny the strong usage of PostgreSQL by the research and teaching communities, but its $\mathcal{CM}$s are ad-hoc and -coded (*C-programs*). Then, their utilization is tedious and time consuming in order to find all formulas. Moreover, as CMs proposed by PostgreSQL run only on PostgreSQL architecture, they can not be used outside (e.g. for NoSQL architecture).

There has been a lot of work vastly explored adapting database $\mathcal{CM}$s. But neither the database community's attention in automatize the process of $\mathcal{CM}$s building. Despite numerous existing $\mathcal{CM}$s only few ones can be used and re-used easily. Indeed, the use of these $\mathcal{CM}$s for performance analysis is still expensive since it requires a good design expertise and being up to date knowledge. Therefore, we notice the automated process via a workflow system using MDE paradigm to take benefits from the model abstraction, the code generation and the reuse capabilities.

6 Conclusion

This article addressed the problem of collaborative development of database cost models. We have presented a *CF-CM* framework to automatize process of cost model development. It covers the fundamental features required by an infrastructure for calibrating such cost model, summarized in the four pillars of cost model editor, code generator service, sharing/collaboration and repository administration. Our solution showed that it is possible to work online collaboratively on cost models entities based on CDO. A Prototype implementation demonstrate the feasibility and practical utility of the approach.

Currently, we are testing our tool by *PhD students* in order to get their feedbacks for possible improvements. Another ongoing work pursues adapting our framework to cover multiple database system types, more precisely NoSQL. It also opens up opportunities, for instance, in online teaching (MOOCs, etc.) by students following our Advanced Databases course for training building cost models and to evaluate the usability and usefulness of our solution.

References

1. D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 466–475. IEEE, 2007.
2. S. Agrawal, S. Chaudhuri, and V. Narasayya. Materialized view and index selection tool for microsoft sql server 2000. *ACM SIGMOD Record*, 30(2):608, 2001.

3. A. Asperti, L. Padovani, C. S. Coen, F. Guidi, and I. Schena. Mathematical knowledge management in helm. *Ann. Math. Artif. Intell.*, 38(1-3):27–46, 2003.
4. D. Bausch, I. Petrov, and A. Buchmann. Making cost-based query optimization asymmetry-aware. In *DaMoN*, pages 24–32. ACM, 2012.
5. T. Burns and other. Reference model for dbms standardization. *SIGMOD Record*, 15(1):19–58, 1986.
6. S. Chacon and J. Hamano. Pro git, vol. 288. *Apress, Berkeley*, 2009.
7. S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*, pages 3–14. VLDB Endowment, 2007.
8. M. Fischer and other. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32. IEEE, 2003.
9. O. M. Group. Omg mof 2 xmi mapping specification. Version 2.4.1, `http://www.omg.org/spec/XMI/2.4.1/`, 2011 (accessed 06.03.17).
10. O. M. Group. Eclipse. the connected data objects model repository (cdo) project. Version 2.4.1, `http://eclipse.org/cdo`, 2012 (accessed 06.03.17).
11. C. He and G. Mussbacher. Model-driven engineering and elicitation techniques: A systematic literature review. In *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016*, pages 180–189, 2016.
12. S. Jean, L. Bellatreche, et al. Ontodbench: Interactively benchmarking ontology storage in a database. In *ER*, pages 499–503, 2013.
13. S. Kent. Model driven language engineering. *Electr. Notes Theor. Comput. Sci.*, 72(4):6, 2003.
14. V. Leis and other. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.
15. C. Maier, D. Dash, I. Alagiannis, A. Ailamaki, and T. Heinis. Parinda: an interactive physical designer for postgresql. In *EDBT*, pages 701–704. ACM, 2010.
16. S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*, pages 191–202, 2002.
17. P. McBrien, N. Rizopoulos, and A. C. Smith. SQOWL: type inference in an RDBMS. In *ER*, pages 362–376, 2010.
18. OMG. Object Constraint Language. Omg available specification. Version 2.0, `www.omg.org/spec/OCL/2.0/`, 2006 (accessed 06.04.16).
19. A. Ouared, Y. Ouhammou, and L. Bellatreche. Costdl: a cost models description language for performance metrics in database. In *Proceedings of the 21ST IEEE ICECCS*. IEEE, 2016.
20. A. Roukh, L. Bellatreche, A. Boukorca, and S. Bouarar. Eco-dmw: Eco-design methodology for data warehouses. In *ACM DOLAP*, pages 1–10. ACM, 2015.
21. A. Roukh, L. Bellatreche, and C. Ordonez. Enerquery: energy-aware query processing. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 2465–2468. ACM, 2016.
22. P. G. Selinger, M. M. Astrahan, et al. Access path selection in a relational database management system. In *ACM SIGMOD*, pages 23–34. ACM, 1979.
23. R. Varadarajan and other. Dbdesigner: A customizable physical design tool for vertica analytic database. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1084–1095. IEEE, 2014.
24. Z. Xu, Y. Tu, and X. Wang. PET: reducing database energy cost via query optimization. *PVLDB*, 5(12):1954–1957, 2012.