Go Meta of Learned Cost Models: On the Power of Abstraction

Abdelkader Ouared[a], Moussa Amrani[b] and Pierre-Yves Schobbens[c]
Faculty of Computer Science / NaDI, University of Namur, Rue Grandgagnage, Namur 5000, Belgium

Keywords: Learned Database Cost Model, Database Management System, Meta-Learning, Generic Interface.

Abstract: Cost-based optimization is a promising paradigm that relies on execution queries to enable fast and efficient execution reached by the database cost model (CM) during query processing/optimization. While a few database management systems (DBMS) already have support for mathematical CMs, developing such a CMs embedded or hard-coded for any DBMS remains a challenging and error-prone task. A generic interface must support a wide range of DBMS independently of the internal structure used for extending and modifying their signature; be efficient for good responsiveness. We propose a solution that provides a common set of parameters and cost primitives allowing intercepting the signature of the internal cost function and changing its internal parameters and configuration options. Therefore, the power of abstraction allows one to capture the designers/developers intent at a higher level of abstraction and encode expert knowledge of domain-specific transformation in order to construct complex CMs, receiving quick feedback as they calibrate and alter the specifications. Our contribution relies on a generic CM interface supported by Model-Driven Engineering paradigm to create cost functions for database operations as intermediate specifications in which more optimization concerning the performance are delegated by our framework and that can be compiled and executed by the target DBMS. A proof-of-concept prototype is implemented by considering the CM that exists in PostgreSQL optimizer.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

A large amount of DBMS have been proposed and used to store and manage the problem of data deluge in storing and accessing. The DB-Engines[1] Ranking of DBMS shows 395 systems (August 2022). In this context, simulation and optimization of query processing are necessary to ensure QoS attributes (e.g. response time, energy) without having to access the full database and hardware of end-users. These techniques require *mathematical* Cost Models (CMs), in order to be precise and exact. This can be achieved by defining the cost functions based on the parameters of a database system (Ouared et al., 2018). The developers of these DBMS are one of the consumers of the CMs. Consequently, the proposed methods and tools assisting these developers are becoming an important issue for industry, science, and open source communities. To that effect, several studies have been conducted to design and provide database CMs in physical design to resolve many important database management tasks, including selection algorithms for physical structures (e.g. materialized views, horizontal partitioning and indexes), buffer management (e.g. (Bellatreche et al., 2013)); query scheduling (e.g. (Kerkad et al., 2014)) and system sizing (e.g. (Zhang and Others, 2011)). The CM is a performance-critical software that is almost always developed in low-level code. While implementations at this level have come a long way, programmers do not trust them to deliver the same reliable performance. Note that many contributions in the open source DBMS community, like PostgreSQL and MySQL, develop plugins to allow database administrators (DBA) to simulate various physical design features and receive quick feedback on their correctness (e.g. (Pantilimonov et al., 2019; Perron et al., 2019; Han et al., 2021; Wu et al., 2013)).

A database CM allows one to *statically* estimate the cost of a query execution, i.e. without having to *execute* this query on a real database and its underlying hardware. By collecting interesting costs estimations (on performance, etc.), it enables one to explore possible settings and select the best execution plan. However, we have identified some limits regarding computation of performance using learned

[a] https://orcid.org/0000-0003-1837-832X
[b] https://orcid.org/0000-0002-6987-1037
[c] https://orcid.org/0000-0001-8677-4485
[1] https://db-engines.com/en/ranking

Ouared, A., Amrani, M. and Schobbens, P.
Go Meta of Learned Cost Models: On the Power of Abstraction.
DOI: 10.5220/0011665800003402
In *Proceedings of the 11th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2023)*, pages 43-54
ISBN: 978-989-758-633-0; ISSN: 2184-4348
Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

CMs. *First*, with the evolution of hardware and software technologies, learned CMs need to be trained on various hardware and data configurations, resulting in a time-consuming and expensive process that has to be repeated when any part of the environment changes. *Second*, this task requires a deep knowledge related to many aspects: databases, hardware, calibration of hyperparameters for Machine Learning (ML), statistical data to estimate parameters such as *selectivity factors* of join predicates, etc. (Ouared et al., 2018). *Third*, for a database CMs to be extended and modified, developers must be parsing a code inside a query engine of DBMS and hand coding inside the source code distribution of the cost model. Currently, database engines are manually optimizing the database CM for each processor technology, which is a costly and error-prone process. To the best of our knowledge, there are no generic interface management facilities to build database CMs. To address this issue, API (Application Programming Interface) CMs are a promising technique that relies on parameterised signature cost functions that enable flexible extensions and modification of CMs, thereby fully unlocking the potential of cost model engineering.

In this paper we address two main challenges: (i) The Cost Functions should be *agnostic* of the underlying programming language used for DBMS implementation, in order to allow their generic application to DBMS. (ii) Learned Cost Functions should *be trained* on diverse hardware platforms to capture specific weights of the unknown parameters. To provide generic interface management facilities for building database CMs, we make the following proposals. First, we propose the use of a common set of building CMs facilities (e.g. algebraic operation, basic operation, primitive cost, parameters, etc.) that are compatible with a wide range of CMs. Secondly, a CM constructor to describe the shape of cost functions for operators conforming to our metamodel; and learn a set of coefficients of these Cost Functions based on models selected from an ML catalog to support a given workload and hardware. Finally, a database generator so that the generated CMs are able to meet the performance requirements by applying different code transformations.

We implemented our framework as part of the PostgreSQL, providing a tool that allows to browse, edit, and select the existing CMs' parameters. In addition, our proof-of-concept prototype provides an automatic calibration solution to estimate the cost of a query plan. We successfully observed and controlled their cost function signatures with the internal parameters and the unit cost functions related to the data primitives of the used data layout, data indexing, and the chosen algorithm of database operations.

After providing the necessary background database operations and CMs in Section 2, we build our framework in Section 3, and describe its implementation in Section 4. We provide concluding remarks in Section 5.

2 BACKGROUND AND RELATED WORKS

In this section, we briefly review the background and the related work.

2.1 Database Operations

A query Q may be defined as a sequence of relation algebra operations (i.e. selection in relational algebra) (Manegold et al., 2002) $\mathsf{Q} = \langle \mathsf{O}_1, \ldots, \mathsf{O}_n \rangle$. Each operator O_i corresponds to the classical DB operators such as restriction, projection, join, scan, sort, etc. (Manegold et al., 2002), and is defined as a quadruplet $\mathsf{O}_i = (\mathsf{Imp}_i, \mathsf{T}_i, \mathsf{C}_i, \mathsf{ProP}_i)$, where

- Imp_i is the implementation algorithm used for the operator, e.g. a Nested-Loops-Join algorithm may implement a *Join* operation;
- T_i is the set of associated input database objects, such as tables, indexes, materialized views etc.;
- C_i is the set of options used to execute the operation, such as adding the buffer option; and
- ProP_i is the operator's *programming paradigm*, e.g. *Sequential*, *MapReduce*, *etc*. (cf. Figure 1).

The execution of an operation O_i is performed in one, or many phases, during which it manipulates resources like relation, buffer allocations, hash table etc. The variability of database operations and their algorithms appears in how to execute this algorithms depending on the specification required by hardware properties (e.g. disk, CPU, GPU technology). For example, the hash join operation on two relations ($R \bowtie S$) has two phases. In the first phase, it reads each relation, applies a hash function to the input tuples, and hashes tuples into buckets that are written to disk. In the second phase, the first bucket of the relation R is loaded into the buffer pool, and a hash table is built on it. Then, the corresponding bucket of the relation S is read and used to probe the hash table.

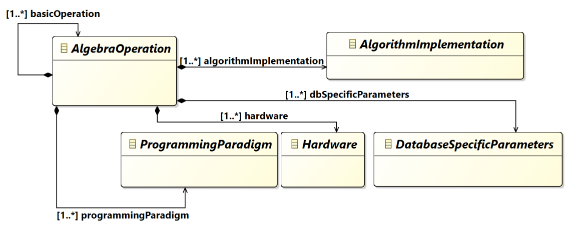

[1..*] basicOperation
AlgebraOperation
[1..*] algorithmImplementation
AlgorithmImplementation
[1..*] dbSpecificParameters
[1..*] hardware
ProgrammingParadigm
Hardware
DatabaseSpecificParameters
[1..*] programmingParadigm

Figure 1: Implementation of Database algebra operation.

2.2 Mathematical Database Cost Models

A QoS attribute is a requirement that specifies criteria that can be used to judge the operation of a system (e.g. databases, operating systems, software). The plan for implementing QoS attributes is detailed in the system architecture and must be captured before the real deployment. Generally, the aspects of QoS considered in the query engines are: (i) *performance*; (ii) *system size*, or *required size*, to the implementation; and (iii) *energy consumption* (Ouared et al., 2018).

A CM is a set of formulas used to estimate the cost of an execution plan. Cost-based query optimizers select the most efficient execution plan based on cost estimations. A CM uses two parts:

Logical Cost. uses *selectivity* and *cardinality* as measures to optimise. The selectivity is the percentage of rows selected by the query, with regards to the total number of rows. The cardinality is the number of rows returned by each operation in an execution plan, which can be derived from the table statistics.

Physical Cost. represents units of work or resource used. Usually, the query optimizer uses disk I/O, CPU usage, and memory usage as units of work.

The CM assigns an estimated cost to any partial or complete plan based on the composition of the physical cost and logical cost. It also determines the estimated size of the data stream for output of every operator in the plan. It relies on the following elements (Chaudhuri, 1998):

1. A set of statistics maintained on relations and indexes, e.g. number of data pages in a relation, number of pages in an index, number of distinct values in a column.
2. Formulas to estimate selectivity of predicates and to project the size of the output data stream for every operator node. For example, the size of the output of a join is estimated by taking the product of the sizes of the two relations and then applying the joint selectivity of all applicable predicates.
3. Formulas to estimate the CPU and I/O costs of query execution for every operator.

EXAMPLE 1 (**CM for PostGreSQL**.) PostGreSQL uses a simple CM that merges the I/O and CPU for different operators O such that its execution cost (i.e., time) can be expressed as:

$$C_0 = n_n.c_n + n_r.c_r + n_t.c_t + n_i.c_i + n_o.c_o$$

where $n_\star$ represent number of pages, and $c_\star$ are coefficients obtained by ML to build regression models for the optimizer CM:

- n_s, n_r , n_t , n_i and n_o are respectively the number of pages sequentially scanned, randomly accessed, cpu tuple cost, cpu index tuple cost and cpu operator cost.
- c_s `seq_page_cost`, the I/O cost to sequentially access a page.
- c_r `random_page_cost`, the I/O cost to randomly access a page.
- c_t `cpu_tuple_cost`, the CPU cost to process a tuple.
- c_i `cpu_index_tuple_cost`, the CPU cost to process a tuple via index access.
- c_o `cpu_operator_cost`, the CPU cost to perform an operation such as hash or aggregation.

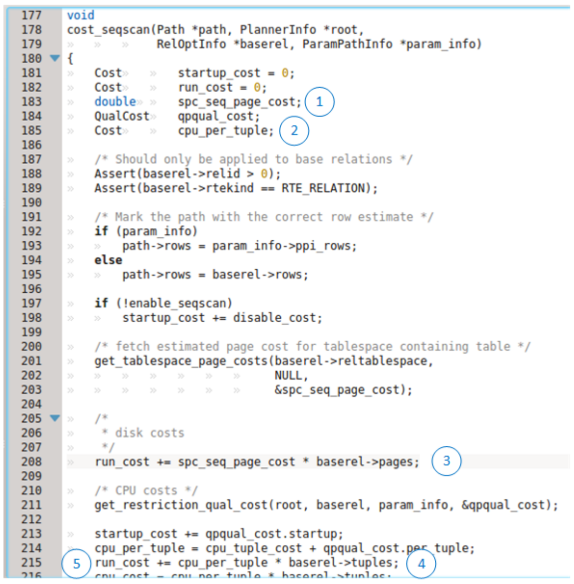
For instance Figure 2 shows the CM of PostgreSQL is available in source file `costsize.c`[2] (Pantilimonov et al., 2019). This file contains procedures to build costs of algebraic operations of any type. It uses a simple CM that merges I/O and CPU for different operators using a fitted model in the form of a linear model. The query optimizer can create an estimate of the overall cost of running queries from its knowledge of individual operator costs, and system parameters. So, if we need to modify the function responsible for creating the CM to extend the cost model. Figure 2 shows a fragment of PostgreSQL CM, the code contains a set of parameters (storage device parameters (cf. ①), processing device parameters (cf. ②)) and a set of primitive cost related to the different steps to execute the *scan operation* (disk cost (cf. ③), CPU cost (cf. ④) and calculating predicate selectivity (cf. ⑤)).

From the example illustrated in Figure 2, the implementation of a database algebra operation depend on three main impact factors:

- **Database-Specific Parameters.** As database-specific factors, we define impacts that are caused by the DBMS that impact the access and the storage of blocks and tuples: the storage model (e.g.

[2]src/backend/optimizer/path/costsize.c

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS



```
void
cost_seqscan(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;

        /* Should only be applied to base relations */
        Assert(baserel->relid > 0);
        Assert(baserel->rtekind == RTE_RELATION);

        /* Mark the path with the correct row estimate */
        if (param_info)
                path->rows = param_info->ppi_rows;
        else
                path->rows = baserel->rows;

        if (!enable_seqscan)
                startup_cost += disable_cost;

        /* fetch estimated page cost for tablespace containing table */
        get_tablespace_page_costs(baserel->reltablespace,
                                                          NULL,
                                                          &spc_seq_page_cost);

        /*
         * disk costs
         */
        run_cost += spc_seq_page_cost * baserel->pages;

        /* CPU costs */
        get_restriction_qual_cost(root, baserel, param_info, &qpqual_cost);

        startup_cost += qpqual_cost.startup;
        cpu_per_tuple = cpu_tuple_cost + qpqual_cost.per_tuple;
        run_cost += cpu_per_tuple * baserel->tuples;
        cpu_cost = cpu_per_tuple * baserel->tuples;
```

Figure 2: Example of C program fragment of PostgreSQL CM.

store data in a row or column oriented way) and the processing model (e.g. tuple-at-a-time, and operator-at-a-time), Page Size and Buffer Management (i.e. page replacement strategies used).

- **Database Operations Algorithms.** the chosen algorithm of a specific database operation influences the performance of database operations. For instance, for join operations, we consider different implementations variants, which are: *nested-loops*, *block-nested-loops*, *hash*, and *sort-merge join*.
- **Hardware Parameters.** changing the processing devices as well as storage devices may influence processing capabilities as well as algorithm design. For instance, using graphics processing units (GPU) as co-processing devices requires to change the join algorithms by considering the parallel execution capabilities of *single instruction multiple data* (SIMD) processors.

2.3 Related Work

Recently, several studies have developed Deep Learning (DL) and Machine Learning (ML) models as opportunities to develop CMs (e.g. (Hilprecht and Binnig, 2022; Ouared et al., 2022; Kipf et al., 2018; Sun and Li, 2019; Ryu and Sung, 2021). In the section of related work, we reviewed the existing languages that aim to leverage Domain-Specific Language Processing for designing database CMs. Similarly, with our work, the first initiatives serve as a starting point for more comprehensive efforts covering aspects of database physical design, optimization, and tuning (e.g. (Bellatreche et al., 2013; Ouared et al., 2016b; Brahimi et al., 2016; Ouared and Kharroubi, 2020)). The work of (Breß et al., 2018) propose a generating custom code for efficient query execution on heterogeneous processors. The same direction is followed in the approach presented in (Wrede and Kuchen, 2020).

Similar efforts have been conducted to make the database CM more generic. These works avoid upgrading and maintaining to rethinking query processing and optimization by adopting the metaphor '*one size fits all*'. Based on this vision, they called *zero-shot learning* for unseen databases using Deep Neural Networks (DNNs) to avoid repeated CM calibration for every new database (e.g. (Kraska et al., 2021; Hilprecht and Binnig, 2022; Hilprecht et al., 2020; Hilprecht and Binnig, 2021)). By exploring the literature, the authors in (Ouared et al., 2016b) propose the first language dedicated to describing database CMs and generate a machinable format. In addition, they propose a repository called *MetricStore* that allows users to upload and download CMs in a structured way, and eases their search and reuse (Ouared et al., 2017).

In DBMS open source, several studies are related to the integration of database CMs inside open source relational DBMS in a way to resolve a specific problem. However, the CM they are hard-coded in the DBMS is not uniform. To avoid such a gap, we must implement a generative and automatic interface to interact with a relational DBMS API using the model-to-text transformation language available within the Eclipse environment. This regular transformation language that may allow the cost function signature to be tailored to a specific application.

3 OUR FRAMEWORK

We propose an MDE-based framework called CoMoRe (**Co**st **Mo**del **Re**finement) that allows the creation of CM cost functions that can be interpreted, and automatically generated for a target DBMS. Figure 6 depicts an overview of CoMoRe, demonstrating a set of generic CM management facilities:

CM Metamodel. Language expresses and controls the CM independently of any DBMS platform, and validating well formedness rules on the cost CMs.

Technical Requirements. consist of a set of design questions relative to aspects of the ML models (e.g. *data distribution*, *variables linearity*, *models type*), and from this information, answering the questions guides users to the kinds of ML models that can address their needs.

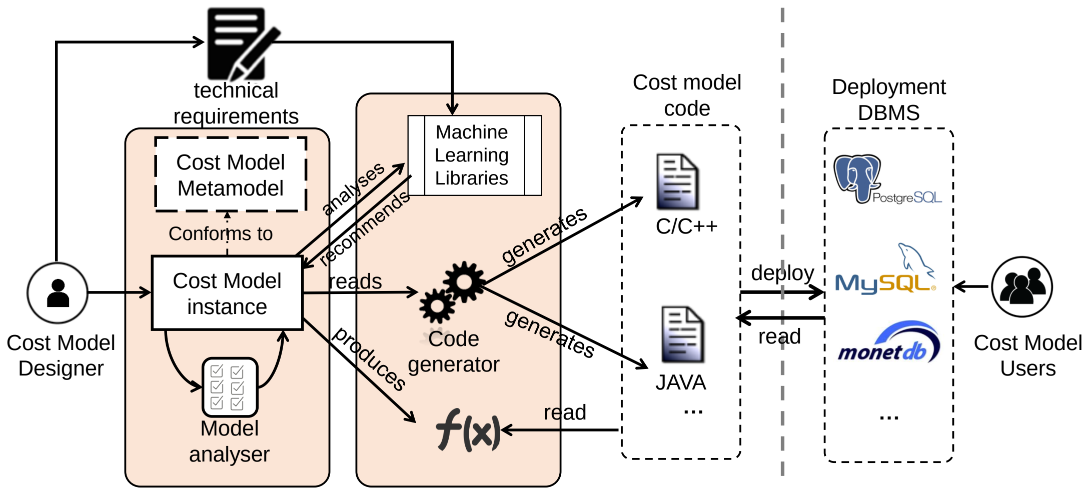


Figure 3: Overview of our framework CoMoRe.

An ML Catalog. that can be used by designers during the designing task of CMs. The selected ML models are used to calibrate the CMs' parameters without developing them from scratch.

Code Generator. In addition, the DSL is complemented with code generators that synthesize CM implementations from CMs for specific requirements. The CM so generated can be deployed in different DBMS platforms (e.g. PostgreSQL or MySQL DBMS) to make them available to users. The generated code via the API that contains parameters with formulas be compiled and executed by the target DBMS (see Fig. 6).

CM designers use our DSL to provide a successful CM product, which can later be exploited by database CM consumers, who are usually database administrators, database analysts, and database architects, to receive quick feedback and solve their problems (e.g., query optimization and physical design, database application deployment over platforms, self-driving the database systems, etc.)

We now present CoMoRe's architecture and available services.

3.1 Layered Design

Modeling the costs in abstraction layers is an important step in achieving our objective. Our work modeling cost queries in multiple layers. It is a useful guideline for our CM building, where different layers of abstraction increase the level of CMs. We begin with a basic CM and then incrementally build this basic model to obtain the global cost.

Database workloads consist of set of queries, and each query is expressed by a set of algebraic operations. We start with an query operation (as shown in Figure 4). At the next layer, these operations provide basic operation with their variants implementation. Finally, we introduce the cost function as composition of the physical cost and logical cost. In this outer layer, operator costs are derived from the characteristics of the system, i.e. *physical cost*) or the cost of various components (CPU, storage disks, and memory) and volume data, i.e. *logical cost* (c.f. Section 2.2).

For example, the cost calculation can be expressed using `SUM` function as follows: ($Cost\ (Op_i) = \sum_{i=1,K}\ ;(o_i)$, $Cost(Query) = \sum_{h=1,m}(Op_h)$, and $Cost(Workload) = \sum_{l=1,n}(Q_l)$).

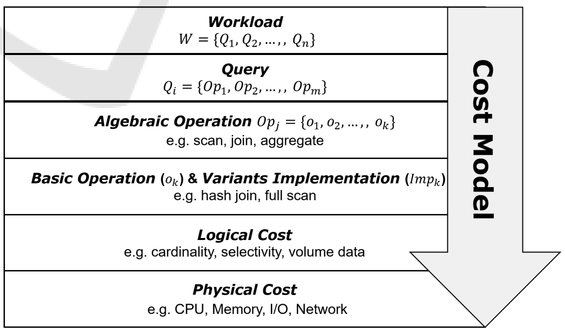

Workload
$W = \{Q_1, Q_2, \dots,, Q_n\}$
Query
$Q_i = \{Op_1, Op_2, \dots,, Op_m\}$
Algebraic Operation $Op_j = \{o_1, o_2, \dots,, o_k\}$
e.g. scan, join, aggregate
Basic Operation (o_k) & Variants Implementation (Imp_k)
e.g. hash join, full scan
Logical Cost
e.g. cardinality, selectivity, volume data
Physical Cost
e.g. CPU, Memory, I/O, Network
Cost Model

Figure 4: Layer abstraction Model.

3.2 Framework Architecture

CoMoRe is an extension of CostDSL (Ouared et al., 2016a), and enables the design of CMs conformant to the metamodel presented in Figure 5. CoMoRe is technically composed of four Ecore packages:

GenericCostModel. This package captures the elements allowing designers to express their CMs.

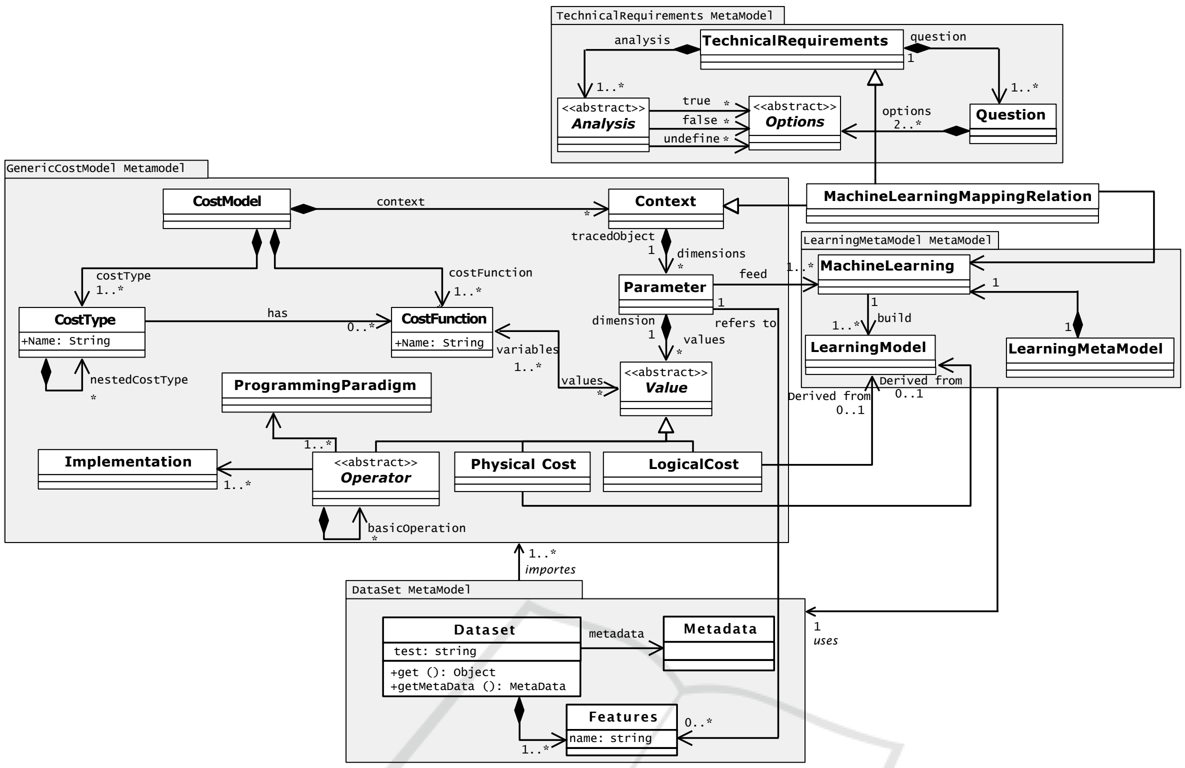


Figure 5: General framework architecture.

Every Context of a given CM is described by a set of database system Parameters. Those Parameters are related to different categories (i.e. Database, Hardware, Query, and Architecture parameters), and help identify data features in a dataset. Thanks to the CostFunction package, the formula of a CM can be expressed using the CM context that allows defining variables of equations.

DataSet. This package provides the test data corresponding to a specific experimental environment including the database, workload, DBMS, devices, platforms, queries, and the quality criteria (e.g. response-time, energy consumption) used to generate these results. Dataset represents the root class that regroups the data features related to a database benchmark. In this case, the value can be generated by a dataset (for example the TPC benchmarks[3]), workload and DBMS. This kind of DataSet of an execution environment is also offered by CoMoRe and is grouped in a category which is dedicated to build CMs.

TechnicalRequirements. CM designers may have difficulties choosing the most suitable ML model that matches the characteristics of the CM under design. TechnicalRequirements helps define different design decisions by giving a precise answer to each question (*true*, *false*, *undefined*) to recommend an appropriate ML model. This concept is connected to CM context and Technical Requirements through the MachineLearningMappingRelation class. This latter is based on constraints that are implemented by using OCL (Object Constraint Language) to provide guidance about what is the ML models can be used.

LearningMetaModel. CoMoRe uses several learning models available in (Hartmann et al., 2019). This package comes with a set of catalogs that represent an organized body of ML models, and is conceived as generalized notions of different learning algorithms available in TensorFlow, Keras, or WEKA. Meta-learning frameworks have been imported to automatically derive these models from certain specifications.

In order to estimate the parameters' values, the metalearning package offers some models (e.g. *linear* and *non-linear multivariate models*) to train a CM. They allow selecting ML models to define the cost function of each CM. The values of the cost function are calibrated by using

[3]http://www.tpc.org/tpch

ML algorithms, with the necessary features coming from the parameters of the CM under design. Furthermore, through the relationship "*derived from*" among `LogicalCost`, `PhysicalCost` and `LearningModel`, one can estimate the cost parameters in the cost function.

3.3 CoMoRe Services

After having presented the architecture of our framework, we can now define the CM services provided by our framework. As before, we consider that the DBMS always has access to its internal CM. In addition, the model DBMS manager is available through its API manager; also, the source code files of the CM are available through the open source DBMS distribution. Figure 6 depicts an excerpt of the interactions between the cost model services and the DBMS optimizer, and the external control panel. We define this generic interface as the set of following services: Intercepting CM Service, Exploratory CM Service, Updating CM Service, Calibrating CM Service and Integrating CM Service.

3.3.1 Intercepting Cost Model Service

This service is invoked when the designer needs to return the list of parameters (i.e. a SELECT statement) and unit cost functions with information such as the parameter definitions. However, the code for the parser can easily be adapted to a CM of other DBMS. For instance, in the case of PostgreSQL, the database CM is stored in two main files: the file `src/backend/optimizer/path/cost.c` and the file `src/backend/optimizer/path/costsize.c`. The file `cost.c` contains function signatures and related definitions for the functions implemented in `costsize.c`. The file `costsize.c` has a function to estimate its cost in terms of the primitive cost variables (*page reads*, *operator evaluation*, etc.), given estimates about the numbers of rows, total data size, etc. We believe that displaying the internal parameters with its cost primitives in structured way helps designers to understand more the signature of CMs to be extended or calibrated.

3.3.2 Exploring Cost Model Service

The presence of a service allowing browsing the existing CM parameters, editing, selecting, predicting, calibrating them, etc. represents a valuable asset for developers/designers. While this service provides an hierarchical taxonomy of concepts indicates hints, informing the designer of any hidden parameters that are not be considered in several CMs. We propose the *Exploring Service* as a solution to provide a common set of parameters and cost primitives allowing intercepting the signature of the internal cost function, and changing its internal parameters and configuration options. For instance, by using this service, designers can display various internal performance parameters like *Shared SQL Pool*, *Redo Log Buffer* and other information concerning the performance.

3.3.3 Updating Cost Model Service

The logic flow in this service follows a set of primitive costs related to the different steps to execute an algebraic operation: (1) estimate the input/output cardinality; (2) compute the CPU cost based on cardinality estimates; (3) estimate the number of accessed pages according to the cardinality estimates; (4) compute the I/O cost based on estimates of accessed pages; (5) compute the total cost as the sum of CPU and I/O cost. Hence, our main task in calibrating is to refine the input/output cardinality for each operator. All these steps need to perform a write during an update operation on a specialized cost formula for each relational algebra operator.

3.3.4 Calibrating Cost Model Service

This service is invoked when the designer needs to calibrate the parameters related to the hardware device. Using technical requirements at the early phase of the design will help perform the right ML model for a specific CM context. This service generates calibrating code to automate the code within programming, the service edits the parameters with its values and lets users do the same thing based on our interface without hand-coding programs. For instance in PostgreSQL DBMS, different types of parameters and configuration options need to be calibrated to update and maintain a CM with each change in hardware devices.

3.3.5 Integrating Cost Model Service

This service is invoked when the developer commits a change to the CM source-code. It handles the target list entries to update or replace existing CM. The list entries are provided with an UPDATE or INSERT statement before or after parsing to modify the server behavior after parsing. Target lists of the element that can be committed are a list of parameters and expressions used in the cost formulas that make the calculation logic of the cost change.

In order to transform any CM conforming to CoMoRe to a target DBMS (PostgreSQL DBMS in our case), we have analyzed 20 well-known DBMS

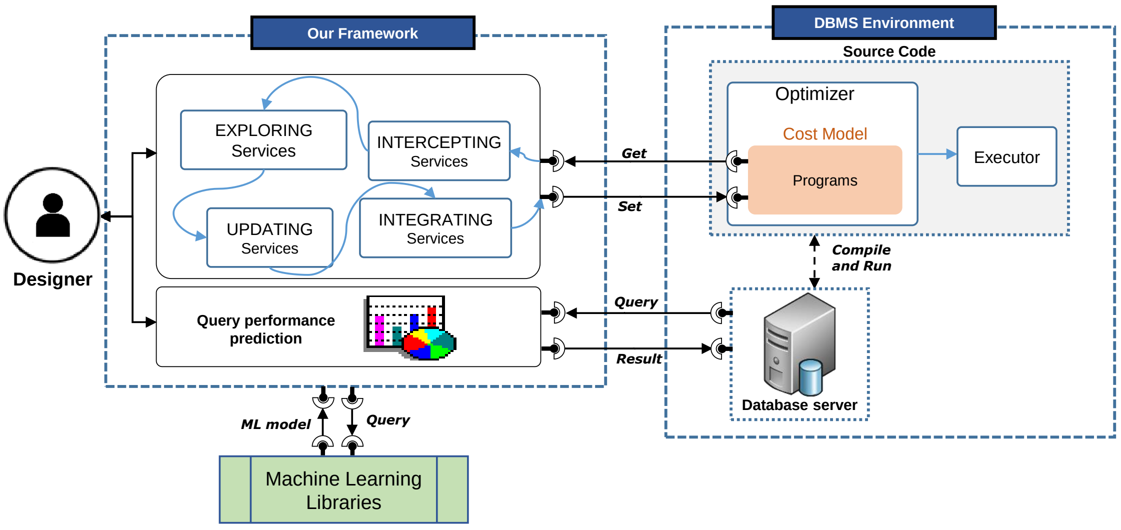


Figure 6: CoMoRe Services.

API and selected the most common structure to enable the management of their internal CMs (Figure 7). CoMoRe can generate code corresponding to the target DBMS thanks to the model-to-text capability provided by the MDE settings. CoMoRe invokes the DBMS via our API to integrate the CM of the database operation under design. Our API provides: **(i)** `Connection` (e.g. PostgreSQL and Oracle), **(ii)** `SQLImplementation` (Native SQL functions and clauses) and **(iii)** `QueryPhysicalPlan` (Specific interpretation of execution plans). For a new RDBMS, we can extended the set of methods (*extractClauses*, *extractTables*, *extractColumns* and *extractParams*) and Execution query Plan (*getTotalCost*, *getTupleNumber*, *getTupleSize*, *getOperations*, *getDuration*, and *extractData*).

Thanks to the MDE interoperability facilities, this service transforms an SQL query instance according to our DSL in the corresponding CM cost function. Note that every CM instance is generated according to the CoMoRe design language. For that, a set of structural rules have been injected in the metamodel. These rules are expressed as OCL invariants. Listing 1 shows an example of OCL structural rule. This rule means that all physical costs and logical costs, which are inputs of a given cost function, have to be referenced as `values` instances in the cost function.

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

Listing 1: An OCL structural rule

```
void
cost_seqscan(Path *path, PlannerInfo *root,
       RelOptInfo *baserel, ParamPathInfo *param_info)
{
  Cost     startup_cost = 0;
  Cost     run_cost = 0;
  Cost     io_cost = 0;
  Cost     cpu_cost = 0;
  Cost     power_cost = 0;
  double     spc_seq_page_cost;
  QualCost   qpqual_cost;
  Cost     cpu_per_tuple;

  /* Should only be applied to base relations */
  Assert(baserel->relid > 0);
  Assert(baserel->rtekind == RTE_RELATION);

  /* Mark the path with the correct row estimate */
  if (param_info)
    path->rows = param_info->ppi_rows;
  else
    path->rows = baserel->rows;
....
}
```

Listing 2: Example of Displaying CM of scan database operation

Table 1 presents some mapping rules between the CoMoRe concepts and the optimizer CMs concepts. Using this mapping, the designer does not have to worry about implementation details. The implementation of the code generator is based on MDE settings using the model-to-text transformation, since our objective is to obtain the cost function of the CM. The implementation relies on the utilization of *Acceleo*. Listing 2 shows the generated CM of scan database operation.

4 PROOF-OF-CONCEPT

CoMoRe is developed as a plugin for Eclipse, based on Java Eclipse Modeling Framework (EMF).

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

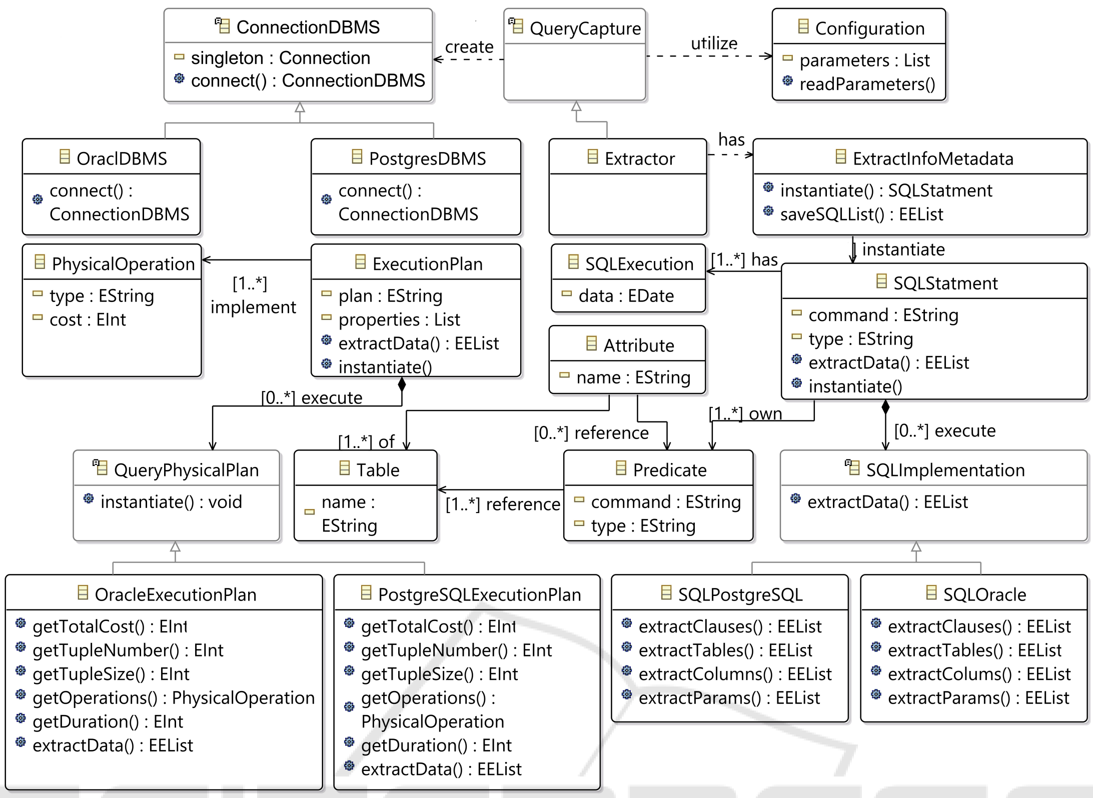


Figure 7: API relationships between our DSL and RDBMS Optimizer.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 1: CoMoRe to RDBMS Optimizer.

CoMoRe Concept	RDBMS API	Comment
Algebra Operation Op	relational algebra operations	one-to-one correspondence
Database	SQLImplementation - $extractClauses()$:List - $extractTable()$:List - $extractColumn()$:List - $extractParams()$:List	Extract statistical values of database (e.g. tuples width, number of rows)
CostModel	PostgreSQLExecutionPlan -$getTotalCost()$:int -$getTupleNumber()$:int, -$getTupleSize()$:int -$getOperations$ ():PhysicalOperation -$getPlaningTime()$:double	Cost values of physical query operators (e.g. I/O cost of the join operation)
QueryPlanner	Execution Plan Implmentation -$instantiate$(plan: String):void -$extractData()$:EElist	$assign_{O_i}(Imp_j, C_j, ProP_i)$: Assign an implementation algorithm , a set of option used as *buffer pool* and execution mode to physical operation O_i

In order to integrate our framework into PostgreSQL to change CM inside the planning query. We implemented CoMoRe as a plugin for PostgreSQL, which extends PostgreSQL (into the flight planning process) allows to designers browse, edit, select, predict, and calibrate the existing CMs' parameters. CoMoRe is composed of two parts, the graphical user interface and the backend contains the DBMS. We based our study on PostgreSQL as it is both open source and based on the C programming language. However, our techniques can be integrated into any other DBMS. Due to the CoMoRe language, the CM of each iteration was modelled and then the code was generated automatically. This process helps to shorten the CM development time. If we would like to do the same stuff with MySQL, it is totally possible we will need to develop a code generator enabling one to transform CoMoRe instances into MySQL programs.

CM Code Inspector

This service shows a visual representation of our metamodel and presents the user guide as a hierarchical taxonomy of CoMoRe vocabulary (Figure 8). In the hierarchy each level describes the components, and the elements of the subordinate-level are instances of elements contained in the superordinate tier. CoMoRe inspects the cost function needed as *function block* to make the CM code more readily usable. Thus, users can display the the cost function as code, signature and components. Since the code structure reflecting the CM domain knowledge. Note that, a cost function may appear in one or more source code fragments (see Figure 9).

PostgreSQL Cost Model Signature

This service shows for each cost function, its shape, its signature and. its components. In addition, users can display and customize various internal performance parameters like the PostgreSQL's standard parameters that can be setting in postgreSQL.conf. The configuration file of PostgreSQL[4] has the default value of the five-cost unit (*cpu_index_tuple_cost*, *seq_page_cost*, *random_page_cost*, *cpu_tuple_cost*, and *cpu_operator_cost*).

CM Calibration

This service provides a model refinement (ML) to predict the output with input parameters to generalize linear/non-linear models. The input parameters are used to learn a CM under design. These parameters relate to the configuration of the initial dataset (e.g. TPC-CH) and configuration of test queries. The best solution is to automatically provide the optimal ML model, by choosing from all possible ML candidates. However, this is a very long and complex task. For the first version of our proof-of-concept prototype, we leave the choice to the user of CoMoRe to decide what fits his/here requirements. As we said before, the catalog systematically organizes ML models. Our implementation uses several different learning algorithms available in WEKA with an alternative library implementation. (Zeileis et al., 2004). The ML model is a pluggable component of our framework, so any other appropriate statistical model can be used.

[4]1 /bin/data folder/posgresql.conf

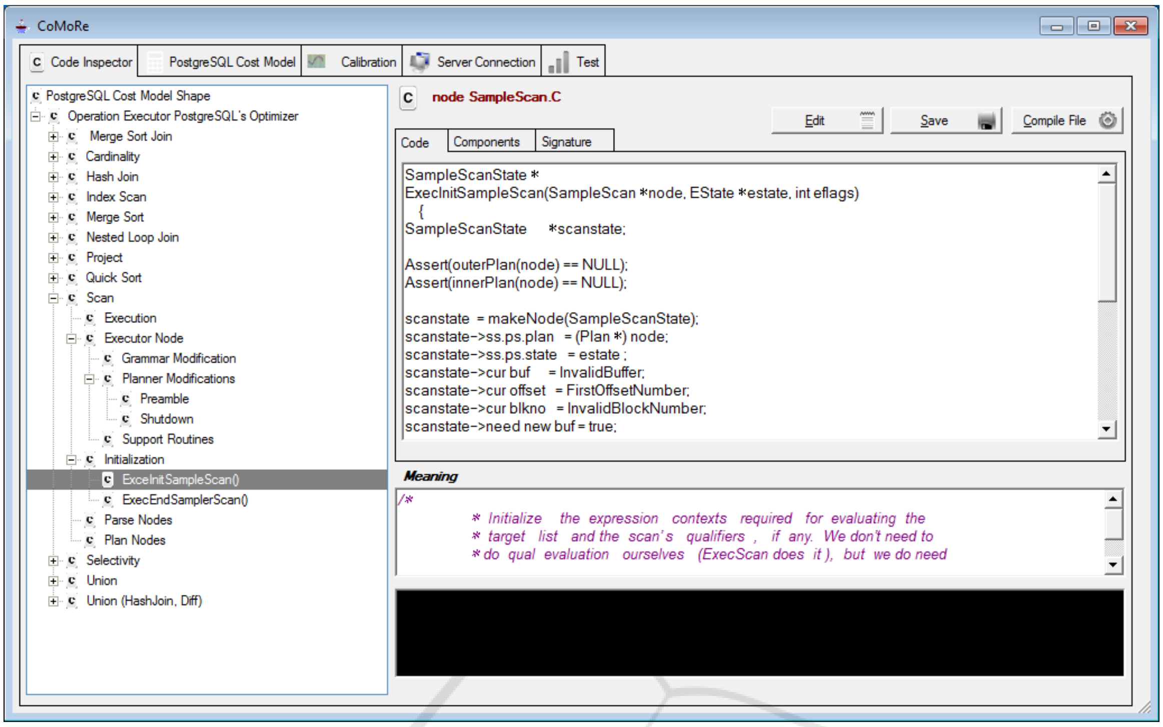
Server Connection and Testing

The server connection responsible for the connection establishment with the DBMS server. When connected, the user has first to select the TPC-H schema database, and evaluate the quality of the database CMs. We relied on mathematical CMs and real DBMS Server across diverse datasets, queries and hardware parameters. After quantitative evaluation, we calculate in terms of the mean relative error ($\mathsf{MRE} = \mathsf{Avg}(\frac{|\mathsf{real}-\mathsf{estimate}|}{\mathsf{real}} \times 100)$) between estimation (i.e. using the mathematical cost model) and real cost (i.e. query execution cost after having access to the full databases and hardware of end-users). The obtained results show the quality of the CM. The desired CM should possess the following features: accuracy, robustness, fast response and portability. If the CM accuracy is acceptable, i.e. have no improvement, the original CM. Otherwise, this process is repeated until requirements are acceptable.

Finally, we can say, with the diversity of database technology, testing in early stages ensures that the designed CM conforms to its requirements and will enable one to explore the performance behavior in terms QoS attributes (e.g. response time, energy consumption) without having to deploy it on a real system, and to compare different alternatives. This solution is in the opposite direction of *Hardware Experimentation* that spends a lot of time/money in testing to compare different alternatives.

5 CONCLUSION

Mathematical database CM is a promising estimation approach for DBMS that enables efficiently execution of the execution of a query. While most open source DBMS already have database CMs, bringing flexible adaptation for a cost model to any relational DBMS is a tedious and error-prone task. A generic solution must support a wide range of relational DBMS independently of the CMs for their implementation, and must be generic to ensure the responsiveness of the CM. We presented a framework based on a generic API, defined independently of any DBMS, and supported by efficient generic interface management facilities. This API is supported by Model-Driven Engineering paradigm to create cost functions for database operations as intermediate specifications that can be interpreted and generated to a target DBMS. Our contributions include an implementation of a prototype tool for the PostgreSQL DBMS. For the first version, we have provided a semi-automated solution to identify the right ML model. However, to fully unlock



- PostgreSQL Cost Model Shape
  - Operation Executor PostgreSQL's Optimizer
    - Merge Sort Join
    - Cardinality
    - Hash Join
    - Index Scan
    - Merge Sort
    - Nested Loop Join
    - Project
    - Quick Sort
    - Scan
      - Execution
      - Executor Node
        - Grammar Modification
        - Planner Modifications
          - Preamble
          - Shutdown
        - Support Routines
      - Initialization
        - ExecInitSampleScan()
        - ExecEndSamplerScan()
      - Parse Nodes
      - Plan Nodes
    - Selectivity
    - Union
    - Union (HashJoin, Diff)

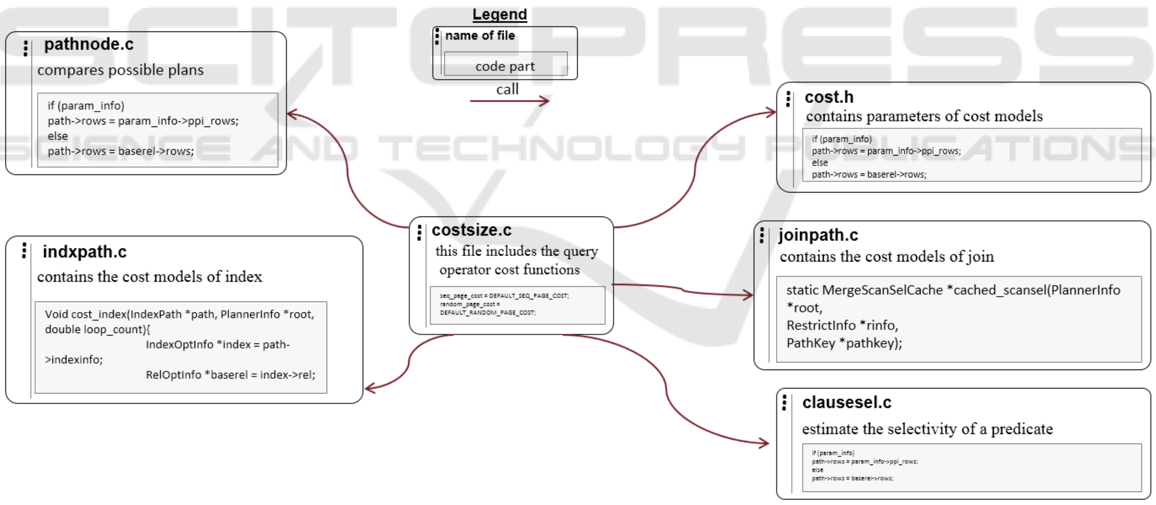
node SampleScan.C

```
SampleScanState *
ExecInitSampleScan(SampleScan *node, EState *estate, int eflags)
  {
SampleScanState     *scanstate;

Assert(outerPlan(node) == NULL);
Assert(innerPlan(node) == NULL);

scanstate  = makeNode(SampleScanState);
scanstate->ss.ps.plan  = (Plan *) node;
scanstate->ss.ps.state  = estate ;
scanstate->cur buf    = InvalidBuffer;
scanstate->cur offset  = FirstOffsetNumber;
scanstate->cur blkno   = InvalidBlockNumber;
scanstate->need new buf = true;
```

```
/*
        * Initialize  the expression  contexts  required  for evaluating  the
        * target  list  and the  scan's  qualifiers ,  if any.  We don't need to
        *do  qual  evaluation  ourselves  (ExecScan does  it ),  but  we do need
```

Figure 8: Proof-of-concept prototype (Screenshots).

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS


Legend
name of file
code part
call
pathnode.c
compares possible plans
indxpath.c
contains the cost models of index
costsize.c
this file includes the query operator cost functions
cost.h
contains parameters of cost models
joinpath.c
contains the cost models of join
clausesel.c
estimate the selectivity of a predicate

```
if (param_info)
path->rows = param_info->ppi_rows;
else
path->rows = baserel->rows;
```

```
Void cost_index(IndexPath *path, PlannerInfo *root,
double loop_count){
                IndexOptInfo *index = path-
>indexinfo;
                RelOptInfo *baserel = index->rel;
```

```
seq_page_cost = DEFAULT_SEQ_PAGE_COST;
random_page_cost =
DEFAULT_RANDOM_PAGE_COST;
```

```
if (param_info)
path->rows = param_info->ppi_rows;
else
path->rows = baserel->rows;
```

```
static MergeScanSelCache *cached_scansel(PlannerInfo
*root,
RestrictInfo *rinfo,
PathKey *pathkey);
```

```
if (param_info)
path->rows = param_info->ppi_rows;
else
path->rows = baserel->rows;
```

Figure 9: Excerpt of relevant files forming default PostgreSQL cost model: The file `costsize.c` contains the source code distribution of the cost model, to calibrate the cost of each relational algebra operator in an execution plan (e.g. *Nested-loops-join algorithm* to implement a Join Operation), developer needs to modify this files and its dependent files (`cost.h`, `pathnode.c`, `indxpath.c`, `clausesel.c`, and `joinpath.c`).

the potential of CM engineering, the ML models must be extracted automatically according to designers' requirements. Future work includes an in-depth study of relationships about a technical requirements and ML models. This will be achieved using a dedicated language to explicitly define the ML model for a specific CM context. We envisage providing an external API to explore the possible internal parameters and configuration options of database CM. Other future work includes addressing some of the limitations of CM services; a usability study would also help to improve the user experience.

REFERENCES

Bellatreche, L., Cheikh, S., Breß, S., Kerkad, A., Boukhorca, A., and Boukhobza, J. (2013). How to exploit the device diversity and database interaction to propose a generic cost model? In *Proceedings of the 17th International Database Engineering & Applications Symposium*, pages 142–147. ACM.

Brahimi, L., Ouhammou, Y., Bellatreche, L., and Ouared, A. (2016). More transparency in testing results: Towards an open collective knowledge base. In *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–6. IEEE.

Breß, S., Köcher, B., Funke, H., Zeuch, S., Rabl, T., and Markl, V. (2018). Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal*, 27(6):797–822.

Chaudhuri, S. (1998). An overview of query optimization in relational systems. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 34–43. ACM.

Han, Y., Wu, Z., Wu, P., Zhu, R., Yang, J., Tan, L. W., Zeng, K., Cong, G., Qin, Y., Pfadler, A., et al. (2021). Cardinality estimation in dbms: A comprehensive benchmark evaluation. *arXiv preprint arXiv:2109.05877*.

Hartmann, T., Moawad, A., Schockaert, C., Fouquet, F., and Le Traon, Y. (2019). Meta-modelling meta-learning. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 300–305. IEEE.

Hilprecht, B. and Binnig, C. (2021). One model to rule them all: towards zero-shot learning for databases. *arXiv preprint arXiv:2105.00642*.

Hilprecht, B. and Binnig, C. (2022). Zero-shot cost models for out-of-the-box learned cost prediction. *arXiv preprint arXiv:2201.00561*.

Hilprecht, B., Binnig, C., Bang, T., El-Hindi, M., Hättasch, B., Khanna, A., Rehrmann, R., Röhm, U., Schmidt, A., Thostrup, L., et al. (2020). Dbms fitting: Why should we learn what we already know? In *CIDR*.

Kerkad, A., Bellatreche, L., Richard, P., Ordonez, C., and Geniet, D. (2014). A query beehive algorithm for data warehouse buffer management and query scheduling. *International Journal of Data Warehousing and Mining (IJDWM)*, 10(3):34–58.

Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P., and Kemper, A. (2018). Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*.

Kraska, T., Alizadeh, M., Beutel, A., Chi, E. H., Ding, J., Kristo, A., Leclerc, G., Madden, S., Mao, H., and Nathan, V. (2021). Sagedb: A learned database system.

Manegold, S., Boncz, P., and Kersten, M. L. (2002). Generic database cost models for hierarchical memory systems. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 191–202. VLDB Endowment.

Ouared, A., Chadli, A., and Daoud, M. A. (2022). Deepcm: Deep neural networks to improve accuracy prediction of database cost models. *Concurrency and Computation: Practice and Experience*, 34(10):e6724.

Ouared, A. and Kharroubi, F. Z. (2020). Moving database cost models from darkness to light. In *International Conference on Smart Applications and Data Analysis*, pages 17–32. Springer.

Ouared, A., Ouhammou, Y., and Bellatreche, L. (2016a). Costdl: a cost models description language for performance metrics in database. In *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 187–190. IEEE.

Ouared, A., Ouhammou, Y., and Bellatreche, L. (2017). Metricstore repository: on the leveraging of performance metrics in databases. In *Proceedings of the Symposium on Applied Computing*, pages 1820–1825.

Ouared, A., Ouhammou, Y., and Bellatreche, L. (2018). Qosmos: Qos metrics management tool suite. *Computer Languages, Systems & Structures*, 54:236–251.

Ouared, A., Ouhammou, Y., and Roukh, A. (2016b). A meta-advisor repository for database physical design. In *International Conference on Model and Data Engineering*, pages 72–87. Springer.

Pantilimonov, M., Buchatskiy, R., Zhuykov, R., Sharygin, E., and Melnik, D. (2019). Machine code caching in postgresql query jit-compiler. In *2019 Ivannikov Memorial Workshop (IVMEM)*, pages 18–25. IEEE.

Perron, M., Shang, Z., Kraska, T., and Stonebraker, M. (2019). How i learned to stop worrying and love re-optimization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1758–1761. IEEE.

Ryu, J. and Sung, H. (2021). Metatune: Meta-learning based cost model for fast and efficient auto-tuning frameworks. *arXiv preprint arXiv:2102.04199*.

Sun, J. and Li, G. (2019). An end-to-end learning-based cost estimator. *arXiv preprint arXiv:1906.02560*.

Wrede, F. and Kuchen, H. (2020). Towards high-performance code generation for multi-gpu clusters based on a domain-specific language for algorithmic skeletons. *International Journal of Parallel Programming*, 48(4):713–728.

Wu, W., Chi, Y., Hacígümüş, H., and Naughton, J. F. (2013). Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment*, 6(10):925–936.

Zeileis, A., Hornik, K., Smola, A., and Karatzoglou, A. (2004). kernlab-an s4 package for kernel methods in r. *Journal of statistical software*, 11(9):1–20.

Zhang, N. and Others (2011). Towards cost-effective storage provisioning for dbmss. *Proceedings of the VLDB Endowment*, 5(4):274–285.