The Generalized Physical Design Problem in Data Warehousing Environment: Towards a Generic Cost Model

Ladjel Bellatreche
LIAS/ISAE-ENSMA
Futuroscope, Poitiers, France
bellatreche@ensma.fr

Sebastian Breß
University of Magdeburg
D-39016, Germany
bress@iti.cs.uni-magdeburg.de

Amira Kerkad, Ahcène Boukorca, Cheikh Salmi
LIAS/ISAE-ENSMA
Futuroscope, Poitiers, France
firstname.lastname@lisi.ensma.fr

***Abstract*—Over the years, plenty of cost models for physical database design were developed. However, solving the physical design problem in a generic way is still a hard task. In this paper, we (1) summarize the development stages of cost models for database systems, (2) propose a generic cost model for the generalized physical design problem, and (3) apply our model on the joint query scheduling and buffer management problem.**

I. Introduction

The physical design phase got more attention from the database community when query optimizers became sophisticated enough to cope with *complex decision support queries* running on extremely large database such as *data warehouses* and large scientific databases. During this phase, the database administrator (DBA) has to select optimization techniques such as materialized views, indexes, or partitioning modes [12]. Two types of optimization structure selection are distinguished: (1) *isolated selection,* where the DBA selects only one optimization technique and (2) *multiple selection,* where several techniques may be selected. Several studies concentrated mainly on the isolated selection. In this case, the physical design problem can be formalized using three components: (1) A workload $W = \{Q_1, Q_2, \ldots, Q_m\}$ containing a set of queries, (2) a set of *storage structures* $ST = \{ST_1, \ldots, ST_n\}$ that can be used by the DBA during the physical design, and constraints related to these techniques and hardware (e.g., storage and maintenance cost, or maximal number of partitions). The problem of the physical design consists of two sub-problems: First, selecting a schema of optimization techniques that reduces the cost of executing a workload W; while second, fulfilling the constraints. Several studies showed the hardness of instances of this problem (e.g., materialized view selection problem).

By exploring the literature, we observe that cost models are well present to quantify the cost of solutions proposed by the aforementioned algorithms. This importance of cost models is not new, because they were used in query optimization, where rule based approaches were substituted by cost based approaches. Cost models needed to be developed to estimate the cost of each relational operation as well as whole query plans.

Cost models can also be used to measure the quality of an optimization technique. Several metrics have been considered by cost models such as CPU time, number of input/output (I/O), and network transfer cost. Usually, I/O is the most frequently used metric.

One of the main characteristics of cost models is that they follow the evolution of database technologies. In the *first generation*, cost models were simple, because they estimated only the cost of relation operations. If we view a cost model as a function, in this generation, a cost model had only two components: a query and a database schema. In the *second generation*, these estimations were enriched by considering optimization techniques such as indexes, and materialized views. The inputs of a cost model were: queries, a database schema and optimization structures. *Third generation* cost models included, the deployment architecture of the database such as distributed or parallel database systems, database clusters, or cloud environments. Inputs were: queries, a database, and the nature of the architecture. With the development of storage models dedicated to databases (e.g., column stores), the *fourth generation* proposed cost models including these storage models. Figure 2 summarizes the evolution of cost models. To the best of our knowledge, existing cost models do not consider all layers (database schema, optimization structures, queries, deployment architecture, storage model) as depicted in figure 1. We argue that in order to develop a generic cost model for physical design, all these layers should be included.

Therefore, the contribution of the paper is as follows: (1) We show the development of cost models. (2) We propose a generic cost model for database systems for physical design problems. (3) We formulate all parameters of our generic cost model. (4) We apply our approach to the joint query scheduling and buffer management (QSBM) problem in the context of relational data warehouses to show applicability. Note that our approach can be used for *any* physical design problem, e.g., partitioning, or index selection.

The paper is structures as follows. We introduce our generic cost model in Section II and we instantiate our

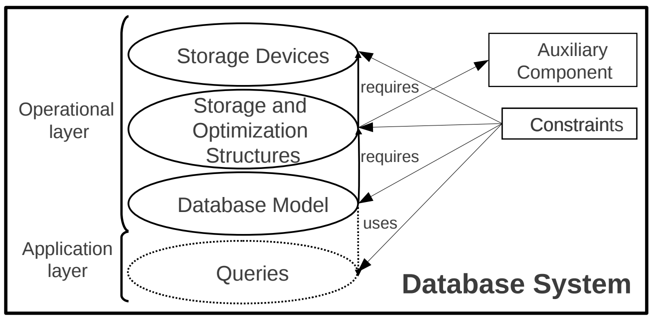

Figure 1. Layers addressed by Cost Models

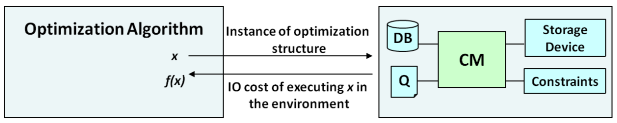
Optimization Algorithm
x
f(x)
Instance of optimization structure
IO cost of executing x in the environment
DB
Q
CM
Storage Device
Constraints

Figure 3. The use of Generic Cost Model in optimization algorithms

generic model for the joint QSBM problem in Section III to prove applicability. Section IV discussed the related works. The paper concludes in Section V.

II. Cost Model Architecture

In this section, we present our generic cost model for the physical design problem, where we highlight the need of use of meta model to represent different components of our cost models.

A. Motivation

To create a generic model for a data warehouse system, we have to take into account several aspects, namely different storage devices, storage structures and database models.

Different Storage Devices: The primary used storage devices nowadays are Hard Disc Drives (HDDs) and Solid State Drives (SSDs). SSDs provide better performance for random access patterns than HDDs, but SSDs are much more expensive [21]. Hence, it is a trade off which storage device is used in the system. Therefore, our meta model has to be able to express characteristics for different storage devices.

Different Storage Structures: The used storage structures have a high impact on database query performance, because they have a large influence on the number of pages that has to be read from disc to process a query. Even plain tables can be stored differently, e.g., row oriented [10] or column oriented [1]. Furthermore, query performance can be accelerated using precomputed results like materialized views [11] or alternative access paths using index structures, e.g., B-tree (one-dimensional) [5] or R/R^*-tree (multi-dimensional) [16], [6]. Each of this storage structures have different performance characteristics. Hence, we have to take them into account in our meta cost model.

Different Database Models/Operation Sets: Nowadays, the most used database model is the relational model introduced by Codd [13]. However, over the years, new database models, and therefore different operations on data sets were developed, e.g., the object-oriented database model [4] or semistructured data models like XML. To cope with this variety of database models, we have to develop a coding for database models in our meta model.

Existing studies are specific to storage devices, storage structures and assume certain database models. Therefore, we need to have a generic model for database systems, in order to create a generic algorithm, which solves the general physical design problem. Our approach is to build a generic cost model, which we will refer to as the *meta model* in the remainder of this paper. The presence of the meta model forces us to develop several types of cost models taking into account the implementation of the storage structures.

B. Layer design meta model

Query: The query is expressed by a formalism which depends closely on the type of database used. However we can represent all queries with different syntax in one meta model. The query has an identifier and a textual description formalism which characterizes the language of the query. The query takes as input a set of concepts which are used to perform a set of relational algebra operations (join, union, etc.) or data manipulation operations (update, insert,etc). An algebra operation can be an unary or a binary function. The result of the query can be restricted by a set of predicates using logical and arithmetic operators or textual operators [27], [22]. The query meta model can easily be extracted from the Common Warehouse Metamodel (www.omg.org/cwm/). Due to the lack of space, we are unable to present all meta models.

Database: We describe an overview of the database meta model. The database is composed of instances which are constituted of tables, indexes, triggers, views and procedures. Tables consist of one or more columns. For each column we must determinate the size, the type of data, the name and the constraints if they exist. The indexes aim at organizing the data of one column on the storage device for minimizing the time of data access. The role of the trigger is the control of data integrity. The stored procedures are a set of SQL statements configured, pre-compiled, stored and executed on demand by the DBMS [25].

Database architecture: There are many deployment architectures of the database as centralized, distributed, parallel or cloud architecture. Centralized architecture has the characteristic that all the data of the database is stored on one storage device. In the distributed architecture, the data is stored in many storage devices in different sites which are connected by the network. When the data is identical in the different sites of the distributed architecture, it is called replicated architecture. The parallel architecture

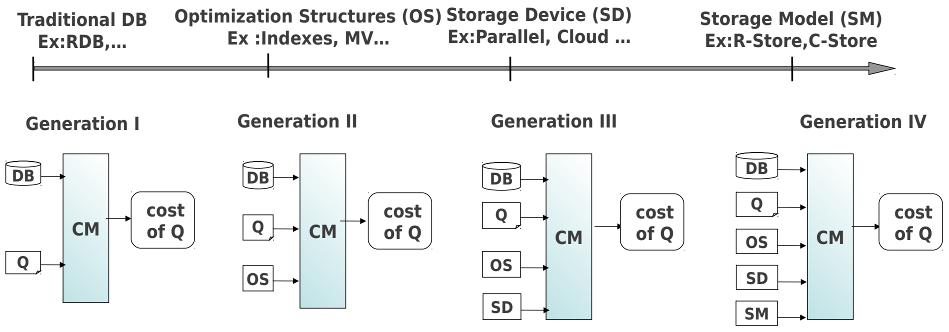

Figure 2. The evolution of Cost Models in Physical Design Problem

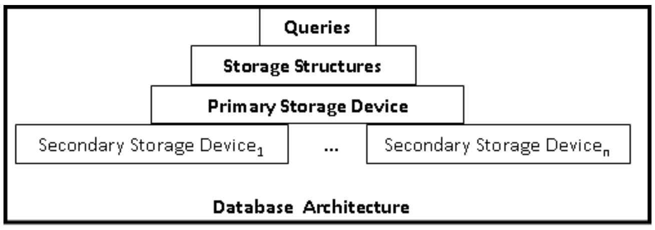

Figure 4. Components of Meta Model

has the same characteristics as the distributed architecture, except that the storage devices are in the same site with many CPU.

Storage Model: There are two storage methods of tables in the storage device, the row-stores and the column-stores. In the first method, the database system stores the table in a sequence of rows, the benefit of this method is easier to insert and to update. In the second method, the database system stores the table in a sequence of columns, the benefit of this method is that any of the column can be as an index and the database system affects only the column to be read during the selection [2].

C. The Generic Cost Model (GCM)

In this section, we propose a generic cost model to describe costs in arbitrary database systems. This means that all existing cost models can be unified in our meta model. To model a database system, one need three components. First, we need the properties of the storage device such as HDD, SSD, main memory, or Cloud Storage. Second, we need to consider the database components, which can be modeled as a set of storage structures $\{ST_1, \ldots, ST_n\}$. Third, we need an abstraction of queries on the database system. A *storage device* is a 5-tuple: SD (L_{read},L_{write},B_{read},B_{write},$size$)

where L_{read}, L_{write} quantify the latency fetching/writing the first page of a data stream ; B_{read}, B_{write} specify the read/write bandwidth of the storage device ; $size$ quantifies the total storage capacity of the device. We differentiate between read and write operations to be able to model storage devices with asymmetric properties, e.g., SSDs.

A *storage structure* ST abstracts how data is managed in a system and is stored on pages $P_1 \ldots P_n$. We denote by $|ST|$ the number of pages of ST.

We define the following functions on a storage structure.

$$rs(ST) = \begin{cases} 1, & \text{if } ST \text{ is row-store} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

$$cs(ST) = \begin{cases} 1, & \text{if } ST \text{ is column-store} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

$$comp(ST) = \begin{cases} 2, & \text{if } ST \text{ is heavy compressed} \\ 1, & \text{if } ST \text{ is lightly compressed} \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

The function K(SD, ST) returns the percentage of pages that are dormant in the database buffer located on the primary storage device, e.g., main memory.

The *operation set* OS: $\{O_1, \cdots O_m\}$ is the set of all operations. Each $O_i \in OS$ can be an unary or a binary function. An unary function gets ST_x as input and returns ST_y as output. Accordingly, a binary function gets ST_x, ST_y as input and returns ST_z as output. Hence, a *query* is a tree of operations O_i with $O_i \in OS$.

Since we abstract from storage devices, we need cost functions to estimate the time to fetch a part of a storage structure in the buffer from storage device SD. One possible approach would be to construct analytical cost models for each instance of a (storage devices, storage structure) pair. However, this would introduce high effort in model maintenance and is in contrast to a generic model that can express everything. Hence, we choose a learning based approach to estimate execution times like [9], [3], [20].

We call $T_{read}(ST_, SD)$ and $T_{write}(ST_, SD)$ the loading and writing time of all pages of the storage structure ST in the storage device SD.

$$T_{read}(ST_, SD) = L_{read} + |ST| \cdot page_{size} \cdot B_{read} \tag{4}$$
$$T_{write}(ST_, SD) = L_{write} + |ST| \cdot page_{size} \cdot B_{write} \tag{5}$$

here $|ST|$ is the number of page of the storage structure $ST_.$ Note that $|ST|$ has to be estimated using storage structure dependent cost functions. We are now able to formally introduce the Database System (DBS) as a tuple

$$DBS = (\{ST_1, \ldots, ST_n\}, OS, SD, \{SD_1, ..., SD_n\}) \quad (6)$$

where $\{ST_1, \ldots, ST_n\}$ is a set of storage structures, OS is a set of operations, SD is the primary storage device where the database buffer is located and the set $\{SD_1, \ldots, SD_n\}$ represents the secondary storage devices. A primary storage device represents the working memory of a database system and a secondary storage device has a persistent memory, larger storage and slower access speed than a primary storage device. Figure 4 summarizes the different "bricks" involved in our generic cost model. With this representation, we can model different types of databases such as in-memory database, database with raid system, database with shared discs.

For each pair (ST, Q_i) we get a sequence $\{O_1, \ldots, O_m\}$ of operations which may vary depending on the chosen optimizations. In particular, for two different optimization choices, the order in which the operations are executed may differ.

We now discuss the cost formulas to estimate I/O cost. The I/O cost of a query Q is the sum of the I/O cost of each operation $O_i \in Q$.

$$IOCost(Q, ST, SD) = \Sigma_{i=1}^{m} IOCost(O_i, ST_i, SD) \quad (7)$$

where ST_i is a set of storage structures resulting from the $i-1$ previous operations. Possible values of ST_i are

$$ST_i = \begin{cases} \{\emptyset, \{ST_x\} & \text{if } O_i \text{ is unary} \\ \{ST_x, ST_y\} & \text{if } O_i \text{ is binary}\} \end{cases} \quad (8)$$

Finally, we need to estimate the cost of executing operation O_i on the storage structure ST_i. We distinguish between row-stores and column-stores:

$$IOCost(O_i, ST_i, SD) = rs(ST) \cdot IOCost_{row}(O_i, ST_i, SD) \quad (9)$$
$$+ cs(ST) \cdot IOCost_{col}(O_i, ST_i, SD)$$

where $IOCost_{row}(O_i, ST_i, SD)$ and $IOCost_{col}(O_i, ST_i, SD)$ are straightforward from the general $IOCost(O_i, ST_i, SD)$ computation formula. The following formulas compute the number of pages $|ST_i|$ of a storage structure:

$$|ST_i| = \frac{\#rows(ST_i) \cdot avgSize(c_i)}{pagsize} \quad (10)$$

for column stores, where c_i is the column involved in ST_i used by operation O_i, and $avgSize(c_i)$ is the average size of the values in column c_i and

$$|ST_i| = \frac{\#rows(ST_i) \cdot avgSize(r_i)}{pagsize} \quad (11)$$

for row stores, where $avgSize(r_i)$ is the average size of a row r_i.

Symbol	Description
ST	storage structure
$P_i(ST)$	ith page of ST
OS	set of operations $\{O_1, \cdots, O_n\}$
W	set of queries $\{Q_1, \ldots, Q_n\}$
SD	storage device
L_{read}	read latency of storage device
L_{write}	write latency of storage device
B_{read}	read bandwidth of storage device
B_{write}	write bandwidth of storage device
$size$	size of storage structure
CM	cost model

Table I
SUMMARY OF NOTATION

D. Examples

We instantiate our model for two scenarios with two common open source database management systems, namely Postgres [24] and MonetDB [7].

Postgres: In the first scenario, we assume that the system has a HDD as secondary storage device. Postgres supports two main index structures: the B^+ Index and the Hash Index[1]. Since Postgres is a row-store, the function *rs(Table)* is true for all tables and *cs(Table)* is always false. Postgres does support compression[2], so the function *comp(Table)* can either return true or false, but the (default) B^+ Index and Hash Index do not support compression. Therefore:

$$\begin{aligned} Postgres = (&\{Table, HashIndex, B^+Index\}, \\ &\{\sigma, \pi, \bowtie, \cup, \cap, groupby, sort, aggregate\}, \\ &MainMemory, \{HardDiscDrive\}) \end{aligned}$$

with

$$comp(HashIndex) = comp(B^+Index) = 0$$
$$rs(Table) = 1 \text{ and } cs(Table) = 0$$

MonetDB: In the second scenario, we assume that the system has a SSD as secondary storage device. MonetDB is a column store, where each column is stored as a Binary Association Table (BAT) [7]. Since MonetDB support compression techniques, the function $comp(BAT)$ can return either one or zero. Therefore

$$\begin{aligned} MonetDB = (&\{BAT\}, \\ &\{\sigma, \pi, \bowtie, \cup, \cap, groupby, sort, aggregate\}, \\ &MainMemory, \{SolidStateDrive\}\}) \end{aligned}$$

with $rs(BAT) = 0$ and $cs(BAT) = 1$

[1]We are aware of extensions like Gist or GIN. However, adding more index structures, although easily applicable, would increase the example complexity.

[2]http://www.postgresql.org/docs/current/static/storage-toast.html

III. Application of the GCM : BMQS problem

As an application example, we propose to instantiate our GCM in a hard optimization problem in physical design. This problem combines the buffer management problem (*BMP*) with the query scheduling (*QSP*) to optimize workloads. In this section, we will give the formalization of the joint problem, namely *Buffer Management and Query Scheduling (BMQS)* problem, followed by an instantiation of our GCM to deal with BMQS.

A. Formalization of the BMQS problem

In this work, we consider some assumptions: **(1)** prior knowledge of the workload (offline scheduling), **(2)** a centralized $\mathcal{RDW}$ environment and **(3)** the initial cache content is considered as empty.

To represent our workload and facilitate handling queries and buffer objects, we use the representation proposed by Sellis et al. by merging all query plans in one graph called Multi View Processing Plan (*MVPP*) [23].

To facilitate the understanding of the formalization of the combined problem *BMQS*, we start by presenting a separate formalization of both *BMP*and *QSP*.

BMP is formalized as follows: **(1)** Inputs : (i) $\mathcal{RDW}$, (ii) a workload with a set of queries $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$ represented by a MVPP, (iii) a set $\mathcal{N} = \{no_1, no_2, ..., no_l\}$ of intermediate nodes of the MVPP candidates for caching, **(2)** Constraint: a buffer size $\mathcal{B}$ and **(3)** Output : a buffer management strategy $\mathcal{BM}$ that allocates nodes in the buffer to optimize the cost of processing $\mathcal{Q}$.

QSP is formalized as follows: **(1)** Inputs : (i) $\mathcal{RDW}$, (ii) a workload with a set of queries $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$, (iii) a buffer management strategy $\mathcal{B}M$. **(2)** Output : scheduled queries $\mathcal{QS} = \{SQ_1, SQ_2, \ldots, SQ_n\}$.

BMQS is described based on the above formalizations as follows:

- Inputs : (i) A $\mathcal{RDW}$, (ii) a set of queries $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$ represented by a MVPP, (iii) a set $\mathcal{N} = \{no_1, no_2, ..., no_l\}$ of intermediate nodes of the MVPP candidates for caching ;
- Constraint: a buffer size $\mathcal{B}$;
- Output : (i) scheduled queries $\mathcal{SQ} = \{SQ_1, SQ_2, ..., SQ_n\}$ and (ii) a buffer management strategy $\mathcal{B}M$, minimizing the overall processing cost of Q.

The *BMQS* problem is NP-hard [14], [17] and impact seriously the performance of overall data warehouse systems. As a consequence, the development of efficient solutions becomes a crucial issue. This efficiency is measured by a cost model, including different layers.

B. Used Cost Model

In order to instantiate our database system (DBS) in the proposed meta model, we describe each component of our system. (1) The storage structure ST used (database component) is relational tables with Row-Oriented Storage. We ignore indexes and materialized views, and suppose there is no data compression. (2) The operation set OS is the set of algebraic operations, e.g projection, selection, join, aggregation, sort, union, which represents the query component in our GCM. (3) The Primary storage device is the main memory where the buffer lies, and the secondary storage device is the Hard Disc Drive. Therefore, our DBS is modeled as :

$$DBS = (\{Tables\}, \{\sigma, \pi, \bowtie, \cup, \cap, groupby, sort, aggregate\},$$
$$MainMemory, \{HardDiscDrive\})$$

$$\begin{aligned} comp(Tables) &= 0 \\ rs(Tables) &= 1 \\ cs(Tables) &= 0 \end{aligned}$$

To instantiate our cost model, we have to estimate the size of intermediate results. However, the number of pages to fetch from disc depends on the buffer content. Therefore, we need to check whether some results (pages) already exist in the buffer. In case some pages are in the buffer, they may be read with much lower latency and higher bandwidth, making T_{read} and T_{write} negligible. Thus, if R_i pages are dormant in the buffer then $T_{read} = T_{write} = 0$. Otherwise, the number of fetched pages is estimated as in [17].

C. Queen-Bee algorithm

As we said before, the joint problem of BMQS is NP-hard. Existing techniques are based on greedy algorithms, costly heuristics etc. Other faster algorithms, which are affinity based, exist but are less efficient. To get a trade-off between speed and efficiency, we propose a divide and conquer algorithm based on queries interaction.

The idea is to group queries by affinity, and to run locally (inside each group of queries) the scheduling and the buffer management strategy. We call these groups "hives", and inside each hive, a query is chosen to be executed first.

This elected query is the "Queen-Bee" of its hive, and it allows filling the buffer with objects that satisfies queries in the same hive. Remaining queries are executed in an order depending on the buffer content. The choice of the queen-bee is done by minimal cost. The next queries of the same hive are ordered by minimal cost considering buffer content.

The algorithm takes a workload to be optimized represented by the merged query plans ($MVPP$) [26]- and starts by generating the query graph with connected components (QGCC), where connected vertices are interacting queries, and each vertex is tagged by its execution cost. The edges are tagged by the number of shared operations (Figure 5)

An optional step is sorting the components, depending on whether queries have priority or not, or if some quality-of-service constraints are given to avoid making some queries

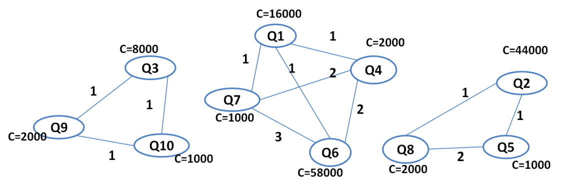

Figure 5. An example of QGCC

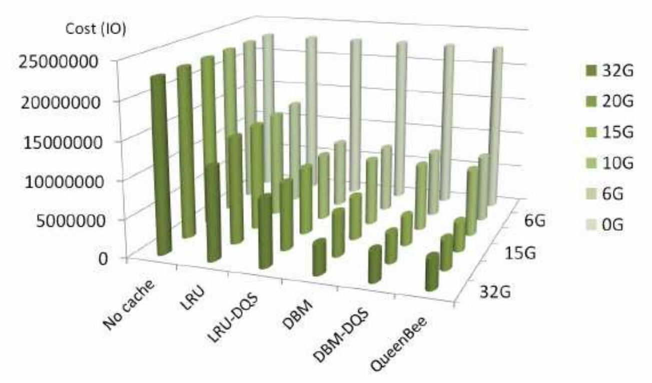

Figure 6. Simulation results using the cost model

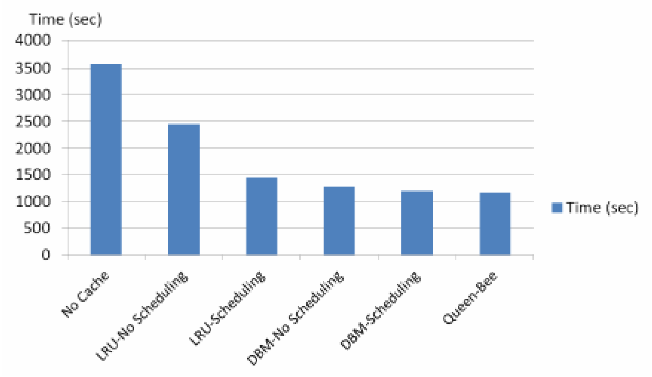

Figure 7. Real costs obtained by deploying simulation results on a DBMS

wait a long time. The final step schedules queries inside each component by traversing its queries. Contrary to the scheduling strategy proposed in [8], called Dynamic Query Scheduler (DQS), that takes into account only the cache content, our scheduler considers other parameters regarding the query execution cost, nodes frequency etc.

The cost model is the kernel of the algorithm because it allows to decide on the order of queries.

D. *Experimental Study*

To show the efficiency of the Queen-Bee algorithm, an experiment is done on Star Schema Benchmark of 100 GB and 30 queries. An instance of the GCM adapted to the dataset and the environment is used.

Our algorithm is compared with 5 baselines : (1) No cache representing initial cost; (2) LRU policy; (3) combining LRU with DQS; (4) a dynamic buffer management policy (DBM); and finally (5) a genetic algorithm using DBM and DQS. Varying Buffer size gives different I/O costs. The results show the high efficiency of the Queen-Bee compared to simplistic and greedy algorithms, as depicted in Figure 6.

To validate the simulation results, the same dataset is considered on a Server (32GB of RAM, 2x2.4GHz of CPU). The used DBMS is Oracle11g and the buffer size is set at 10GB. Figure 7 shows the real execution cost in seconds. In addition to the efficiency of Queen-Bee, the validation shows the similarity between simulation cost estimated by our cost model, and the real cost. This proves the quality of our GCM which is able to consider different environment parameters.

IV. Related Work

The cost models are considered an important part of query optimizers and physical design selection algorithms. With the spectacular interests that the database community gave to the physical design in the 90's, developing algorithms for selecting optimization techniques became a necessity. Note that physical design represents one of the hardest problems for database management systems [18]. To satisfy this requirement, several studies proposed algorithms dealing with one instance of the physical design with simple cost models. They take into account database parameters (tables' size, attributes' length, etc.) and disk parameters (e.g. disk page size) [26], [15]. These cost models have been extended by taking into account the different deployment architectures of database applications (distributed, parallel, database cluster, and recently cloud). Afterwards, with development of advanced devices such as flash, researchers proposed cost models including the characteristics of these devices in their components. Some research studies considered other auxiliary aspects related to the DBMS components, such as buffer management. Manegold et al. propose a framework [19], which allows to automatically create cost functions of database operations for each layer of the memory hierarchy by combining the memory access patterns of database operations. Our approach is designed to be able to address all physical design problems, whereas the approach of Manegold et al. is tailor made for estimating database operations cost during query processing in a generic way. Hence, they do not focus on the general physical design problem, as we do. One of the major differences despite that, is that Manegold et al. differentiate between sequential and random access latency and bandwidth, whereas we consider different values for read/write latency and bandwidth to model storage devices. We see a possible link of our approach and of Manegold et al. for future work, because our approaches can complement one another, e.g., the framework of Manegold et al. could deliver cost metrics for database operations on a system, whereas our model can utilize this cost metrics to build our generic cost model for a given database system.

V. Conclusions and Outlook

In this paper, we showed the great impact of cost models along the different generations of databases. Usually, they are largely used in the context of the physical design phase of extremely large databases. Recall that physical design represents one of the hardest problems for database management systems. This importance was amplified by the birth of decision support applications. In this paper, we establish a clear discussion about the cost model evolution and the different layers that have to be considered in their development. We identify four layers : (1) Queries, (2) Database, (3) Database Architecture, and (4) Storage Model. Based on this discussion, we propose a generic cost model and provide an example of its instantiation. To make the connection between our cost model and the physical design problem, we consider one exiting problem by combining two well known NP-hard optimization problems, namely buffer management and query scheduling. This joint problem is solved by using an algorithm inspired from bee life and using our generic cost model. A theoretical validation using our mathematical cost model is given and the obtained results are validated on Oracle11g. The results are encouraging.

References

[1] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented Database Systems. In *VLDB*, pages 1664–1665. VLDB Endowment, 2009.

[2] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD*, pages 967–980. ACM, 2008.

[3] M. Akdere and U. Çetintemel. Learning-based Query Performance Modeling and Prediction. In *ICDE*, pages 390–401. IEEE, 2012.

[4] M. P. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik. The Object-Oriented Database System Manifesto. In *SIGMOD*, pages 395–408. ACM, 1990.

[5] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. In *SIGFIDET*, pages 107–141, 1970.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, pages 322–331. ACM, 1990.

[7] P. Boncz and M. L. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *BIWIT Workshop*, pages 240–251. Computer Society Press, 1995.

[8] L. Bouganim, F. Fabret, P. Valduriez, and C. Mohan. Dynamic query scheduling in data integration systems. In *ICDE*, pages 425–435. IEEE, 2000.

[9] S. Breß, F. Beier, H. Rauhe, E. Schallehn, K.-U. Sattler, and G. Saake. Automatic Selection of Processing Units for Coprocessing in Databases. In *ADBIS*, pages 57–70.

[10] D. D. Chamberlin, M. M. Astrahan, and M. W. e. a. Blasgen. A History and Evaluation of System R. In *Commun. ACM*, volume 24, pages 632–646. ACM, 1981.

[11] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing Queries With Materialized Views. In *ICDE*, pages 190–200. IEEE, 1995.

[12] S. Chaudhuri and V. R. Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*, pages 3–14. VLDB Endowment, 2007.

[13] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.

[14] A. Diwan, S. Sudarshan, and D. Thomas. Scheduling and Caching in Multi-Query Optimization. In *COMAD*, pages 150–153. Computer Society of India, 2006.

[15] H. Gupta. *Selection and maintenance of views in a data warehouse*. Thesis, Stanford University, 1999.

[16] A. Guttman. R-Trees - A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 47–57. ACM, 1984.

[17] A. Kerkad, L. Bellatreche, and D. Geniet. Queen-bee: Query interaction-aware for buffer allocation and scheduling problem. In *DaWaK*, pages 156–167, 2012.

[18] W. Labio, D. Quass, and B. Adelberg. Physical database design for data warehouses. In *ICDE*, pages 277–288, 1997.

[19] S. Manegold, P. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB*, pages 191–202. VLDB Endowment, 2002.

[20] A. Matsunaga and J. A. B. Fortes. On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications. In *CCGRID*, pages 495–504. IEEE, 2010.

[21] M. Polte, J. Simsa, and G. Gibson. Comparing Performance of Solid State Devices and Mechanical Disks. In *PDSW*, pages 1–7, 2008.

[22] M. A. Roth, H. F. Korth, and D. S. Batory. Sql/nf: A query language for 1nf relational databases. *Information Systems*, 12(1):99 – 114, 1987.

[23] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[24] M. Stonebraker and L. A. Rowe. The Design of POSTGRES. In *SIGMOD*, pages 340–355. ACM, 1986.

[25] T. J. Teorey, S. S. Lightstone, T. Nadeau, and H. Jagadish. *Database Modeling and Design:Logical Design*. University of Michigan, fifth edition, 2011.

[26] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *VLDB*, pages 136–145, 1997.

[27] M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324 –343, 1977.