Automatic Physical Design Tuning based on Hypothetical Plans

Ana Carolina Almeida[1], **Angelo Brayner**[2], **José Maria Monteiro**[2], **Sérgio Lifschitz**[3], **Rafael Pereira de Oliveira**[3]

[1] Universidade do Estado do Rio de Janeiro - UERJ
ana.almeida@ime.uerj.br

[2]Universidade Federal do Ceará - UFC
{brayner,monteiro}@dc.ufc.br

[3]Pontifícia Universidade Católica do Rio de Janeiro - PUC-Rio
{sergio,rpoliveira}@inf.puc-rio.br

Abstract. *It is well-known that fine tuning in database physical design is an important strategy for speeding up data access. In this paper, we introduce a new approach, denoted HypoPlans, to make relational database systems able to execute self-tuning actions, based on the notion of Hypothetical Query Execution Plans. HypoPlans is non-intrusive and completely autonomous. In this sense, it is DBMS-independent and does not require any DBA intervention. Our approach is based on heuristics that run continuously. Thus, HypoPlans is able to guide decisions on the current physical database configuration in order to dynamically react to workload changes. More specifically, we present in this paper the software architecture of a framework, which implements HypoPlans. In order to evaluate the viability of our approach, we have instantiated this framework for the database physical design concerning index (self)tuning. Our experiments show that HypoPlans is quite effective and efficient, also presenting low resource consumption.*

1. Introduction

Database applications have become very complex, dealing with a huge volume of data and database objects (e.g., tables, indexes and materialized views). Concurrently, low query response time and high transaction throughput have emerged as mandatory requirements to be ensured by database management systems (DBMSs). Among other possible interventions regarding database performance, database physical design fine tuning has become a critical task since it may considerably speed up data access. However, manual adjustment for these large systems may be unfeasible in many practical situations. Consequently, several approaches for reducing human intervention in database physical design tuning activity have been proposed [Bruno 2011].

We classify database physical design tuning tools as *(i)* continuous or non-continuous; *(ii)* autonomous or non-autonomous; and *(iii)* intrusive or non-intrusive. Such tools are characterized as *continuous* or *non-continuous* depending whether or not they are able to dynamically capture and analyze the current workload. *Autonomous* physical design tuning tools have the ability of automatically triggering adjustments in database physical design (creation, dropping or rebuilding of database structures such as index and materialized views). Conversely, non-autonomous tools transfer to the DBA this responsibility.

Intrusive solutions are those that require changes and that are tightly coupled to a particular DBMS code.

In this work, we present *HypoPlans*, which is a continuous, autonomous and non-intrusive approach for physical design maintenance. *HypoPlans* may be characterized by the following features: *(i)* it can be used with different DBMSs in a plug-and-play manner; *(ii)* it is based on hypothetical query execution plans for dealing with automatic physical design maintenance, and; *(iii)* there is a low-overhead during the physical design tuning activity, since *HypoPlans* captures workloads in a lightweight manner.

The rest of this paper is organized as follows. Section 2 discusses some automated database physical design tuning tools and strategies. Section 3 presents the *HypoPlans* approach and software architecture, besides the notion of hypothetical plans. Section 4 brings the description of a *HypoPlans* instantiation and analyzes some experimental results. Section 5 concludes this work.

2. Related Work

Bruno *et al* [Bruno 2011] present an intrusive index tuning tool implemented as a component of Microsoft SQL Server. In [Maier et al. 2010, Alagiannis et al. 2010] an intrusive and interactive solution, called PARINDA (PARtition and INDex Advisor), is presented. Bruno et al propose in [Bruno and Chaudhuri 2010] an intrusive and interactive tool for Microsoft's SQL Server, which is similar to PARINDA since it makes tuning sessions interactive, allowing DBAs to try different tuning options and interactively obtain a feedback. A new index recommendation technique, called semi-automatic index tuning, based on the notion of a *work function* algorithm was proposed in [Schnaitter and Polyzotis 2012]. In [Narasayya and Syamala 2010], the authors describes a non-continuous approach to solve the problem of identifying the set of indexes that has to be defragmented for a given workload. The proposed solution was implemented in an intrusive manner, specifically Microsoft's SQL Server. It is important to highlight that, all these works presented in this section adopt an intrusive approach.

Regarding industrial tuning tools, most of them are intrusive, non-continuous and not fully autonomous, such as DB2 Advisor, Database Tuning Advisor (MS SQL Server) and SQL Adjust Advisor (Oracle) [Bruno 2011]. Existing continuous and autonomous tools are all intrusive [Bruno 2011].

3. HypoPlans: hypothetical plans for a non-intrusive approach

In our approach, called *HypoPlans*, a database structure (e.g. an index or a materialized view) may be either real or hypothetical. A *real* structure exists physically, i.e., it occupies disk space and may be used for accessing data. On the other hand, for the *hypothetical* structure, its definition exists only in the *HypoPlan*s' catalog. Indeed, the hypothetical structures are used only for external *what-if* evaluations.

HypoPlans adopts the classical observation-prediction-reaction phases for self-tuning [Weikum et al. 1994]. During the *observation phase*, *HypoPlans* monitors and analyzes each task, in the workload (see Definition 1), in order to identify the most adequate database structures and their respective benefits for that task (see Definition **??**). Throughout the execution of the *prediction phase*, *HypoPlans* attempts to infer the effects that would result from changing the current physical design configuration C to a new configuration $\bar{C}$, where $\bar{C}$ contains both real and hypothetical structures. The *reaction phase* has the functionality of physically changing C to $\bar{C}$.

Definition 1 (Workload) *Let T_i be a task represented by a triple $<$ Q, P, CE_P $>$, where Q is the SQL expression of query Q, P represents the execution plan P used to process Q and CE_P corresponds to estimated cost to execute plan P. We define a workload W for a given DBMS in an instant t as the set $W = \{T_1, T_2, ...T_n\}$ of tasks.* $\diamond$

Definition 2 (Benefit) *Let T be a task of a workload W ($T \in W$), i a database structure (an index ou a materialized view, for example), $cost(T)$ the cost for executing T without using i and $cost(T, i)$ the execution cost using i. The benefit of using i to execute T, denoted $B_{i,T}$, is computed as follows: $B_{i,T} = max\{0, cost(T) - cost(T, i)\}$.* $\diamond$

HypoPlans implements the new concept of *hypothetical query execution plans*. The key idea behind the concept of hypothetical query execution plan is to enable *HypoPlans* to substitute a given sub-plan p belonging to the original execution plan P by a hypothetical sub-plan p', obtained through the use of hypothetical structures. *HypoPlans* can build a hypothetical plan HP, which is equivalent to the original plan P, but with a lower estimated execution cost. Thus, the hypothetical structures in a hypothetical sub-plan p' are good candidates to be physically created. In order to compare the execution costs of HP and P, a Canonical Cost Model (*CCM*) is used just for supporting *HypoPlans* in finding efficient physical design tuning activities.

Definition 3 (Hypothetical Execution Plan) *A hypothetical query execution plan (for short, hypothetical plan) HP for a given task T is a query execution plan with the following features: (i) the database structures (indexes or materialized views) used in HP may be real or hypothetical; (ii) it is built from a real execution plan P (the execution plan used by the DBMS to run the query T), and (iii) it is equivalent to P, i.e., HP and P yield the same result whenever they are applied to execute T.* $\diamond$

Figure 1 depicts an abstract model of *HypoPlans'* architecture. To run *HypoPlans* for a given DBMS, it is necessary to instantiate three drivers: one for workload access (DWA), a driver for statistics access (DSA) and another driver for DBMS update (DDU). These 3 drivers are the only components of *HypoPlans* that are DBMS-specific. Nevertheless, remember that these drivers are implemented in a non-intrusive fashion. The main components of the architecture illustrated in Figure 1 are the following:

Workload Obtainment (WO). The WO component is responsible for periodically capturing the workload submitted to the DBMS. Thus, WO has to access the DBMS catalog in order to get the tasks executed by the DBMS. Thereafter, the triple <*SQL Expression, execution plan, estimated cost*> representing each captured task is stored in a structure, called Local Metabase.

Local Metabase (LM). LM stores the captured workload and a set of data on database structures. For each index structure, for example, *HypoPlans* stores in LM the following data: the structure id, the name of the table on which the structure is defined, the columns that compose the search key, the structure state (real or hypothetical), the index type (i.e., primary or secondary), the estimated cost for creating the structure and its accumulated benefit.

Driver for Workload Access (DWA). This component enables *HypoPlans* to access

metabase (or catalog) of a given DBMS.
Statistics Obtainment (SO). This component is in charge of accessing statistics information of the target DBMS, such as table cardinality, the amount of disk pages required to store a database table, the height of (b^+ tree) index structures and so forth.
Driver for Statistics Access (DSA). This driver is responsible for ensuring the access to database statistic data.
Integrated Heuristics for Index Maintenance (IHIM). A set of heuristics, which encapsulates the knowledge of the index self-tuning activity.
Index Maintenance (IM). The main functionality of IM is to analyze each task T of a workload W in order to identify an appropriate index configuration IC for W. Thus, IM is responsible for executing the observation and prediction phases.
DDL Generator (DG). This component executes the reaction phase. Its key goal is to build DDL commands in SQL for either creating, dropping or rebuilding database structures (indexes and materialized views) into the target DBMS.
Driver for DBMS Update (DDU). It is responsible for enabling the execution of DDL commands (built by the DG component) in the target DBMS.

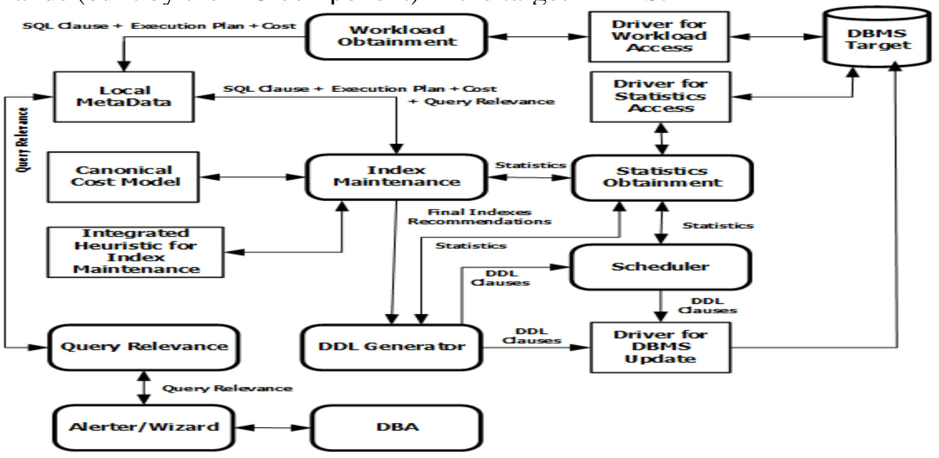

Figure 1. HypoPlans' Architecture.

It is important to make the following observations. First, the CCM does not have the goal of being as precise as the DBMS internal cost model, since a hypothetical query plan will not be used by native DBMS query optimizers. In other words, CCM has to be precise just for supporting *AIM-HypoPlans* in finding a good set of candidate indexes for a given SQL query. Second, the "what-if" approach requires as input parameters an SQL query and a set of candidate indexes in order to provide an efficient execution plan and its estimated execution cost. The "what-if" approach is not able to autonomously define a set of candidate indexes which may reduce the response time for a given task t. Third, inserting or deleting tuples into/from a table T imposes updates to the index structures defined on T. Hence, the cost of updating an index structure is considered a negative benefit. Fourth, *AIM-HypoPlans* considers both primary (clustered) and secondary (non-clustered) indexes.

4. Experimental Results

We have instantiated the proposed approach to solve the problem of automatic index tuning. This implementation, called *AIM-HypoPlans*, is available online[1]. We have imple-

[1]https://github.com/BioBD/dbx

mented *AIM-HypoPlans* (and its drivers) in Java for running it in three different database servers: PostgreSQL 8.4 and two major DBMS. The names of the DBMSs have been omitted for legal reasons and we call these DBMSs and their advisers of A and B.

In order to evaluate *AIM-HypoPlans*, we have executed several experiments using *AIM-HypoPlans* and the advisers A and B. The workload used in our experiments has been defined based on TPC-H benchmark. Out of all TPC-H queries we have chosen six queries, which have the highest response times, namely queries 1,2,4,10,17 and 19. To carry out the tests a 13GB TPC-H database (TPC-H scale factor 10) has been created. We ran the tests using an Athlon 64x2, 2 GHz workstation with 2GB of RAM and a 250GB disk. In order to analyze the *AIM-HypoPlans'* effectiveness, we have defined the following strategy: for each index i recommended and created by *AIM-HypoPlans*, we have identified if the index i was suggested by the DBMS's adviser (A or B), if i has been really used by to process queries submitted to the DBMS (A and B) and if the index i is recommended by the DBT-3 (TPC-H benchmark). The results are presented in Table 1.

Query	Table	Column	Adv. A	Used by A	Adv. B	Used by B	Used by PS	AIM-HP	DBT-3
1	Lineitem	L_shipdate, L_Returnflag, L_linestatus	Y	N	N	N	N	Y	N
1	Lineitem	L_shipdate	N	N	Y	N	N	Y	Y
2	Partsupp	PS_partkey, PS_suppkey, PS_supplycost	Y	Y	N	Y	Y	Y	N
2	Part	P_partkey	Y	N	N	N	Y	Y	Y
2	Partsupp	PS_partkey	N	Y	N	Y	Y	Y	Y
2	Supplier	S_suppkey	N	N	N	N	Y	Y	Y
4	Lineitem	L_orderkey, L_commitdate, L_receipdate	Y	Y	N	Y	Y	Y	N
4	Orders	O_orderdate, O_orderkey, O_orderpriority	Y	Y	N	Y	Y	Y	N
4	Lineitem	L_orderkey	N	Y	Y	Y	Y	Y	Y
4	Orders	O_orderdate	N	Y	Y	Y	Y	Y	Y
10	Customer	C_custkey, C_nationkey, ..., C_comment	Y	Y	N	Y	N	Y	N
10	Lineitem	L_orderkey, L_returnflag	Y	N	N	N	N	Y	N
10	Orders	O_orderdate, O_orderkey, O_custkey	Y	Y	N	Y	Y	Y	N
10	Lineitem	L_orderkey	N	N	Y	N	N	Y	Y
10	Orders	O_orderdate	N	N	Y	N	Y	Y	Y
10	Customer	C_custkey	N	N	N	N	N	Y	Y
17	Lineitem	L_partkey	Y	N	N	Y	Y	Y	Y
17	Part	P_brand, P_container	N	Y	Y	Y	Y	Y	N
19	Lineitem	L_partkey,..., L_shipninstruct	Y	Y	N	Y	N	Y	N
19	Part	P_brand, P_container, P_size, P_partkey	Y	Y	N	Y	Y	Y	N
19	Lineitem	L_partkey	N	Y	Y	Y	N	Y	Y
19	Part	P_brand, P_container	N	N	Y	N	Y	Y	N
19	Part	P_brand, P_container, P_size	N	Y	Y	Y	Y	Y	N

Table 1. ***HypoPlans*** **effectiveness. Yes (Y) or No (N) for index creation.**

Observing Table 1, one can note that *AIM-HypoPlans* has recommended and created the 11 index structures suggested by DBT-3, whereas adviser A has recommended only 2 structures. Moreover, from the set of indexes suggested by adviser A, DBMS A has exploited approximately 64% (7 from 11 indexes). From the set of indexes created by *AIM-HypoPlans*, DBMS A has employed approximately 57% (13 from 23 indexes). *AIM-HypoPlans* has recommended and created the 11 index structures suggested by DBT-3, whereas adviser B has recommended 9 structures. Regarding the eleven index structures recommended by adviser B, *AIM-HypoPlans* has recommended 11 indexes from the 11 suggested by adviser B. One can verify that from the set of indexes suggested by adviser B, DBMS B's query processor has used approximately 55% (5 from 9 indexes). From the set of indexes created by *AIM-HypoPlans*, DBMS B has used approximately 61% (14 from 23 indexes). The query engine of DBMS B has exploited more indexes automatically created by *AIM-HypoPlans* than the indexes recommended by adviser B. In relation to PostgreSQL, from the set of indexes created by *AIM-HypoPlans*, the PostgreSQL's query optimizer has used 15, i.e., a *use ratio* of 65%. Therefore, the CCM implemented by *AIM-HypoPlans* is a good approximation of the DBMS A's internal cost model.

5. Conclusion

In this paper, we have presented an autonomous and non-intrusive approach, *HypoPlans*, for database physical design maintenance in zero-administration environments. *HypoPlans* periodically captures the workload from the database metabase The autonomous behavior ensures two advantages: 1) to dynamically and more quickly react to changes in the workload by making decisions on the current database physical design whenever necessary (without having to wait for the action of a DBA), and; 2) to give tuning suport to companies without a specialized DBA, as well as for environments where DBA intervention proves impractical, which is the case of cloud database environments. A non-intrusive solution is that whose code is loosely coupled to a specific DBMS code, in order to, for example, be able to make calls to the DBMS query optimizer. Thus, a new release of the DBMS query optimizer does not imply in a new code of the self-tuning solution. Nonetheless, *HypoPlans* depends on database metadata. However, we believe that changes in metadata structure is less unlikely than changes DBMS code. Besides, if the DBMS metadata eventually changes, the impact of this change is limited on the proposed approach, since only the drivers should to be updated.

We conclude this paper by emphasizing that *HypoPlans* presents features, which are essential for companies without a specialized DBA, as well as for environments where DBA intervention proves impractical, which is the case for cloud database environments, where autonomy and reliability are key features.

References

Alagiannis, I., Dash, D., Schnaitter, K., Ailamaki, A., and Polyzotis, N. (2010). An automated, yet interactive and portable db designer. In *Proceedings of the 2010 ACM SIGMOD international conference*, SIGMOD '10, pages 1183–1186, New York, NY, USA. ACM.

Bruno, N. (2011). *Automated Physical Database Design and Tuning*. Emerging directions in database systems and applications. CRC Press.

Bruno, N. and Chaudhuri, S. (2010). Interactive physical design tuning. In *International Conference on Data Engineering*, pages 1161–1164.

Maier, C., Dash, D., Alagiannis, I., Ailamaki, A., and Heinis, T. (2010). Parinda: an interactive physical designer for postgresql. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 701–704, New York, NY, USA. ACM.

Narasayya, V. and Syamala, M. (2010). Workload driven index defragmentation. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 497–508. IEEE Computer Society.

Schnaitter, K. and Polyzotis, N. (2012). Semi-automatic index tuning: Keeping dbas in the loop. *Proceedings of the VLDB Endowment*, 5(5):478–489.

Weikum, G., Hasse, C., Moenkeberg, A., and Zabback, P. (1994). The COMFORT automatic tuning project, invited project review. *Information Systems*, 19(5):381–432.