Caching All Plans with Just One Optimizer Call

Debabrata Dash, Ioannis Alagiannis, Cristina Maier, Anastasia Ailamaki

Ecole Polytechnique Fédérale de Lausanne, Switzerland

{debabrata.dash,ioannis.alagiannis,cristina.maier,anastasia.ailamaki}@epfl.ch

Abstract— **Modern database management systems (DBMS) answer a multitude of complex queries on increasingly larger datasets. Given the complexities of the queries and the numerous design features, manual design is no longer an option. Instead, automatically designing the database is vital to maximize its performance and to reduce the total cost of ownership. For this purpose, commercial DBMS feature automated physical designers suggesting an efficient DB design by using the optimizer as a cost model. Unfortunately, consulting the optimizer is time-consuming, an effect which is typically counter-acted by drastically pruning the search space, thereby potentially missing the optimal solution. Recently techniques cache the optimizer's output and evaluate some plans with the cached results, reducing the number of calls to the optimizer. Still, however, the cost of invoking the optimizer to fill the cache is nontrivial, undermining scalability when running workloads with thousands of queries. In this paper, we use the intermediate optimization results in a dynamic programming based optimizer to reduce the cache initialization overhead. We demonstrate the accuracy and efficiency of our techniques by implementing them on the PostgreSQL open source query optimizer. For a star-schema workload, our techniques build the cost model 5 to 10 times faster than the conventional approach, while preserving accuracy.**

I. INTRODUCTION

Modern commercial database management systems typically provide automated physical designers [1][2][3], i.e., tools that suggest a set of physical structures (indexes, partitions, and materialized views) to improve performance of a certain workload. Although implementation details may vary, all designers implement the architecture shown in **Figure 1**: the designer accepts as input the workload as well as a set of constraints, and identifies a set of candidate structures, which can improve the performance of the workload. The designer then invokes the optimizer repeatedly to quantify each candidate structure's benefit. Finally, a complex search algorithm identifies the combination of structures (*configuration*) that provide the maximum benefit, while satisfying the user-defined constraints.

The designer typically produces numerous candidates, the evaluation of which requires substantial disk space and time. To economise run-time resources, modern designers make decisions based on "*what-if*" questions, which can be answered by only creating the statistics of the candidate structures without actually building them. The optimizer only depends on the statistics of the features to determine their benefits; since the statistics can be accurately simulated there is no loss of information, therefore what-if questions significantly improve designer performance and scalability.

Unfortunately, even when asking what-if questions the optimizer still takes a significant amount of time to compute the benefit of each configuration: in a typical physical design process about 90% of the time is spent on invoking the optimizer [4]. To reduce this overhead, designers aggressively prune candidate configurations through greedy algorithms, often reducing the quality of the final solution. Two recently-proposed approaches reduce the overhead of optimizer invocation without aggressive premature pruning: parametric query optimization (C-PQO) [5] and caching partial query plan costs (INUM) [4].

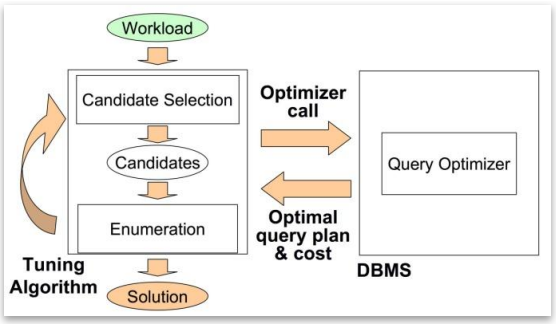

Figure 1 Architecture of a typical physical designer.

C-PQO reuses the parametric query optimization (PQO) infrastructure of SQL Server to optimize a query and cache the union of all feasible execution plans for the given query. In terms of eliminating optimizer-caused overheads C-PQO is currently the state-of-the-art; however, it is built into SQL Server's top-down query optimizer. Therefore, it is not straightforward to port C-PQO to another query optimizer, such as the bottom-up query optimizer of PostgreSQL.

Generally speaking, an ideal approach would drastically reduce the need for costly optimizer calls without binding the process to a certain optimizer's features. Optimizer-independent techniques such as INUM evaluate a carefully chosen representative set of plans for each query, build a plan-cache and determine the costs for all possible configurations using the cached plans. The cache eliminates optimizer calls once it is built; filling the cache, however, still requires a non-negligible number of optimizer calls (92 calls on average for TPC-H queries [4]). Although caching allows for four to five orders of magnitude more configurations to be evaluated and offers higher-quality solutions when compared to the no-caching approach, cache construction costs limit the scalability of INUM. This overhead limits the applicability of INUM's cost model to online workloads, where the caches need to be constructed in the order of milliseconds per query. Lowering the cache construction overhead also helps for off-line designers, where the indexable structures such as materi-

alized views and partitions are created dynamically, since a plan-cache must be built for every query using those structures.

In this paper we look deeply into the plan-caching approach and discover that much of the information generated during the optimization process, if exported to the designer, can drastically reduce the related overhead. Indeed, while evaluating each configuration, the optimizer creates and evaluates several *intermediate plans*, some of which already constitute the answer to subsequent optimizer calls. The intermediate plans are sub-optimal plans for the given configuration, but are optimal for other configurations. If instead of caching only the final plan we also cache the intermediate plans, several of the subsequent calls to the optimizer can be suppressed, minimizing the related overhead. Furthermore, the technique does not significantly compromise the technique's portability and independence, as most optimizers follow a similar evaluation process.

To demonstrate and evaluate the technique we use the dynamic programming-based optimization process of the PostgreSQL open-source DBMS query optimizer. Using INUM as the caching mechanism we implement PINUM, an index selection tool for PostgreSQL. We obtain intermediate plan evaluations piggy-backed to the answer to each what-if question. We choose PostgreSQL because of its relatively mature query optimizer. We select INUM as its interface with the optimizer is lighter than C-PQO, making it more portable across DBMS and their releases. We first implement what-if indexes, then port INUM's cache model to enable scalable candidate space search. By adding a small set of query optimizer hooks, we experimentally find that the additional information reduces INUM's cache-building costs by a factor of 5 to 10. We then integrate the cache-based query cost estimation with a simple index selection tool to suggest indexes that speed up simple analytical queries by factor of 10.

Our focus on PostgreSQL is also motivated by lack of mature automated physical designers for open source DBMSs, although they are relatively mature for commercial DBMSs. Monterio et al. implement and design an index suggestion tool for PostgreSQL [6]. They, however, assume the size of the indexes to be zero, severely affecting the accuracy of the optimizer when using their what-if indexes. Thiem et al. propose a physical designer for Ingres open source DBMS which focuses more on efficient performance monitoring, than physical design [7]. Kao et al. propose changing the optimizer to store access path decisions in a data structure and suggest the frequently requested access paths [8]. This technique, however, cannot be applied to a bottom-up optimizer, which does not request access paths.

The rest of the paper is organized as follows: We provide the necessary background on INUM and PostgreSQL query optimizer in Sections II and III respectively. Section IV analyses the inefficiencies in the current approach, and Section V discusses PINUM's extensions and their implementation on the query optimizer. We discuss the experimental results using PINUM in SectionVI. Finally, Section VII concludes and discusses the future research direction.

II. INUM OVERVIEW

INUM postulates that, although selection tools must examine a large number of alternative designs, the number of different optimal query plans and, thus, the range of different optimizer outputs is much smaller. Therefore, it makes sense to reuse the optimizer output, instead of calling the optimizer to generate plans that differ by just one access path. INUM works by first performing a small number of key optimizer calls per query in a pre-computation phase and caching the optimizer output (query plans along with statistics and costs for the individual operators). During normal operation, query costs are derived exclusively from the pre-computed information without any further optimizer invocation. The derivation involves simple numerical calculations and is significantly faster compared to the complex query optimization code. To explain INUM's postulations, we borrow the following definitions from the literature:

1. *"Configuration"* is a set of indexes. A configuration is called *"atomic"* with respect to a query, if for each table in the query; at most one index is present in the configuration [10].
2. An *"interesting order"* is a tuple ordering specified by the columns in a join, group-by or order-by clause **Error! Reference source not found.**.
3. An *"interesting order combination"* for a query is the set of interesting orders, where there is at most one interesting order for each table involved in the query [4].
4. An index *"covers"* an interesting order, if the interesting order is the first column in the index. Similarly an atomic configuration *covers* an interesting order combination [4].

An interesting order of a table is a column, which, if ordered, reduces the query cost. For instance, in the query "select A, B from T order by A", A is an interesting order for table T. If there are two tables T1 and T2, and their interesting orders are A, C respectively, then possible interesting order combinations covered by a combination are (A,Φ), (A,C), (Φ,C), and (Φ, Φ). We denote lack of interesting order on a table as Φ. The atomic configuration {T1(A),T2(C)}, consisting of indexes on A and C, covers the interesting order combination (A,C). Let D be a non-interesting order column in T2, the atomic configurations (T1(A)) and (T1(A),T2(D)), cover the interesting order combination (A, Φ).

Using these definitions, the most important observations from INUM are:

1. If a query involves only merge- and hash-join plans, the cost of join and aggregation does not depend on the cost of accessing data from the table or indexes. The total cost of the query depends linearly on the cost of accessing data for each table. The cost of accessing data includes the cost of accessing all required rows and columns from indexes, tables, or a combination of them.

2. If a query involves only merge and hash join plans, then caching one plan per interesting order combination is sufficient to find the plans for all possible atomic configurations.

3. For queries involving all join methods including Nested-Loop Joins, it caches more than one plan per interesting order

combination and achieves reasonable cost approximation for the optimal plan cost.

INUM separates the total cost of the query into "internal" join-aggregation costs, and the "leaf" data access costs. The internal costs, determined by join methods and join orders, are only allowed to change between different cached plans. In a given cached plan, the internal cost remains constant, and the variations in the query cost comes from the variation of the data access costs. Therefore, the cost of the plan linearly depends on the data access costs, which allows easy determination of the optimal plans and the optimal cost of a query in presence of atomic configurations.

In the rest of the paper, we denote the data access costs as simply "access costs", and the internal plans along with the join method and orders as the "*INUM cache"*.

Using atomic configurations limits INUM from suggesting plans involving index intersections. Typically the complexity overhead of using intersections outweigh the modest improvement in performance they provide [5]. We therefore focus on improving INUM's performance instead of extending it to include such complex index operations.

III. The PostgreSQL Query Optimizer

In this section, we describe the query optimizer in detail and later we discuss the components we change to implement PINUM on top of PostgreSQL [9].

Figure 2 shows the very high level architecture of the query optimizer. Given a query the workflow in the components is as follows:

The *Query Preprocessor* statically analyses the query and identifies the opportunities to optimize it using rewriting. Then the *Sub-query Planner* optimizes each sub-query that cannot be merged into the top-level query individually. In this step, it identifies the sub-queries and invokes the next component on each of them.

The *Grouping Planner* isolates the grouping and ordering columns. It also identifies the interesting orders for the query. The *Access Path Collector* component iterates over the tables in the from clause, and finds the costs of accessing those using operations such as table scans, index scans, or seeks. It looks up the statistics of the table as well as indexes from the *Catalog* schema and estimates the costs of accessing them. It also attempts to reduce the complexity of next components by eliminating inefficient access paths. If two indexes cover the same interesting order, then this component filters out the access path with the higher cost. This filtering process provides the best access path for each interesting order, therefore preserving the potential of using the interesting order in subsequent steps.

The *Join Planner* component implements a dynamic programming algorithm to identify the join methods and join orders. Given a query joining n relations, the join planner's dynamic program consists of n-1 levels. In the first level, optimal join methods are determined for every two pairs of relations. Every subsequent level adds one more relation to the join of the previous level and finds the optimal plan for the join. The *plan* is a tree of operations with the internal nodes of the tree determining the joining method, and the leaves represent the access paths on indexes and tables. The plan also stores the interesting order it covers. The top level provides a set of optimal plans with different interesting order combinations.

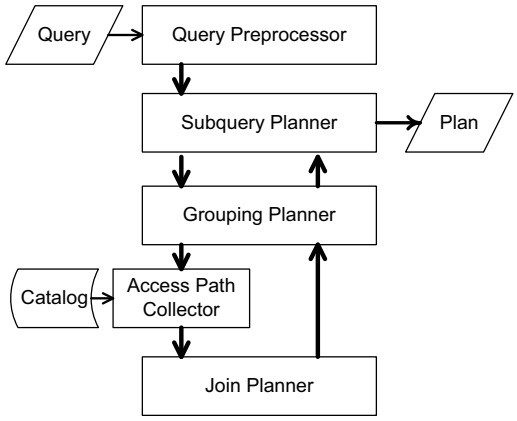

Figure 2 The PostgreSQL optimizer's architecture

On the return path, the grouping planner adds the grouping constructs such as group-by, order-by, distinct etc. to the plans. If the grouping can be done using one of the interesting orders covered by the plan then the plan is forwarded as such, otherwise sort steps are added to provide the required ordering. The sub-query planner then combines all sub-query paths into one path. Finally, it returns the plan to the caller.

The optimizer maintains *intermediate plans* during the join planner, and after returning from the join planner to the grouping planner. These plans join a subset of the queried tables, or lack the required aggregation operations.

IV. Harnessing the intermediate plans

In this section we describe our intuition behind harnessing information from the intermediate plans created by the optimizer towards saving future what-if questions. To illustrate the points, we analyse the INUM plan cache when evaluating TPC-H queries; consider, for instance, the 5th query in the TPC-H benchmark. The query joins 6 tables in the benchmark, and groups and orders the results. Since the join and order-by clauses contribute to the interesting orders, the query has 648 interesting order combinations.

INUM needs to query the optimizer 648 times to fully build the cache; if we carefully parse the plans, however, we find only 64 unique plans in the cache; 90% of the optimizer calls and the cached plans are therefore redundant! Furthermore, even the optimizer call to evaluate a useful interesting order combination finds plans for other interesting orders and discards them before reporting the optimal plan. For example, if the optimizer is invoked with an index set covering the interesting order combination (A,B,C), then the optimizer finds many of the plans providing interesting order combinations (A,Φ, Φ), (Φ,B,Φ) etc. It prunes the plan providing (Φ,B,Φ), only if it costs more than a plan providing a more specific interesting order combination, such as (A,B,Φ). All non-

pruned plans are collected during join optimization, only to be discarded at the final optimization level before reporting the optimal plan.

We can make the INUM cache construction much more efficient by collecting the discarded plans along with their interesting order information. Once a set of plans is collected we determine the next best interesting order combination to send for optimization and greedily fill the entire cache. If we have access to the optimizer code, however, we can do even better: by omitting the pruning process mentioned above, a single call to the optimizer returns the optimal paths for *all* possible interesting order combinations. (If we use INUM we need to request separate plans for when nested-loop joins are disabled, so we need to make two calls.) To be fair, we do introduce a (potentially significant) overhead, as the optimizer builds 648 plans and transfers them to the client. Section V-D describes a pruning technique to reduce this overhead and output only the 64 useful plans.

V. Design and implementation

We first modify the PostgreSQL optimizer to provide the APIs that INUM's cache requires, such as what-if indexes, and optional disabling of nested-loop joins. It then tweaks the optimizer to speed up the cache construction. Finally, it integrates the cache with a simple index selection tool to automatically suggest indexes.

A. What-If Indexes

To determine the optimal plans in presence of an index, the query optimizer uses two types of statistical information – the size of the index, and histograms of the columns in the index. Since the histogram information is associated with the table, we do not replicate or modify them. To compute size, we use the average attribute size, the total number of rows, and the attribute alignments to find the number of leaf pages required to store the index. We ignore the internal pages of the B-Tree index, since they affect the relative page sizes only on very small indexes.

B. Porting INUM to PostgreSQL

As Section 3 describes, INUM needs the index access costs from the optimizer along with the optimal plans for each interesting order combination. Since INUM considers the plans with nested loop joins separately, the optimizer also needs to provide a way to disable nested loop joins.

To disable the nested loop joins, we use the global parameter "enable_nestloop" in the DBMS. Originally, this parameter adds a very high overhead to the nested loop joins, thus discouraging its use. Since PINUM requires the nested loops to be completely absent from the suggested plans, we tweak the join planner to remove nested loop operations if this flag is set.

Getting the access cost is the simpler of the two problems. Naively, the optimizer can be queried with a single index per each table in the query and the access cost can be determined by parsing the generated plan. This process is relatively inefficient since it re-optimizes the entire query to find the access cost for a small set of indexes. What follows is a discussion on how to speed up this process.

C. Speeding up Access Cost Lookups

To speed up index access cost lookups, we modify the access path collector module in the optimizer. Given a large set of what-if indexes, the access path collector finds the access paths for all those indexes and keeps only the least expensive index access path for each interesting order. We modify the module to keep all index access paths, instead of the least expensive one. This allows PINUM to determine the access costs of a large set of indexes by calling the optimizer just once.

D. Speeding up the Cache Construction

INUM caches two optimal plans for each interesting order combination, one with nested loop joins and one without (i.e. containing only hash and merge joins). To find these plans, it first enumerates all combinations of the interesting orders and invokes the optimizer for each one of them with after creating indexes covering those interesting orders.

Instead of using ad-hoc pruning, we reduce the overhead by observing that the join planner keeps at least one path for each interesting order combination. It keeps them with the hope of using the interesting order in a merge join or in the grouping planner. Therefore, if the optimizer is invoked with all possible interesting orders, then the join planner maintains the optimal plans for every useful interesting order combination until the last level. Instead of replicating INUM's plan set inside the optimizer, we prune away unhelpful interesting order combinations by using the following condition: *If plans A and B provide interesting orders in set S_A and S_B, where $S_A \subseteq S_B$ and $Cost(S_A) < Cost(S_B)$, then we remove Plan B.*

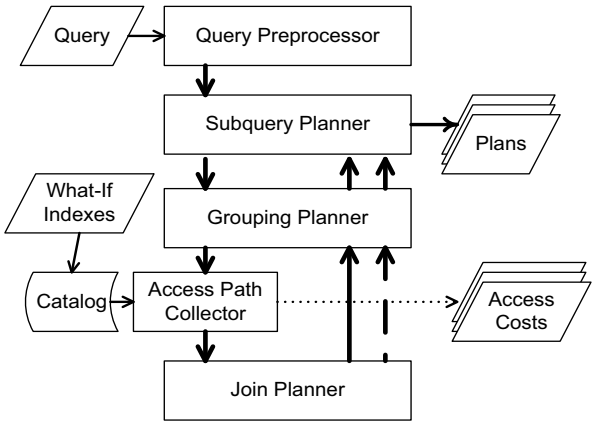

Figure 3 The modified query optimizer architecture.

In other words, if a plan requiring smaller interesting order set is more efficient than a plan requiring large interesting order, then the inefficient plan can be safely removed. This pruning process reduces the search space of the join planner, while preserving all useful plans.

The nested-loop joins are attractive at low access costs, but become expensive as the access cost of the table grows. Therefore, the same interesting order can have multiple optimal plans. Which forces INUM to make multiple calls to the

optimizer to estimate the costs of the plans containing nested-loop joins. Typically, only two calls to the optimizer at the extreme access costs are sufficient to achieve reasonable accuracy. If higher accuracy is required, the pruning condition can be changed to prune only when the access cost range of the indexes are in the same range. This provides higher accuracy, but at the cost of a bigger plan cache and slower cost lookup.

Figure 3 shows the architecture of the query optimizer after the modifications for what-if indexes, fast access cost lookup and fast cache construction are added. The dotted and dashed lines represent the new data flow induced by PINUM's access cost and cache construction optimizations. As the figure shows, the changes to the optimizer components are minimal and requires only touching three files in the optimizer codebase.

E. Integration with an Index Selection Tool

To demonstrate the effectiveness of PINUM's caching model we integrate it with a simple index selection algorithm to create a complete index selection tool. The tool expects a workload and a space budget as input. It determines a set of indexes which occupies less than the budgeted space and attempts to provide the maximum speed up to the workload.

The tool first statically analyses the queries to find a large set of candidate indexes. It then follows an iterative algorithm, and selects the index which provides the most benefit to the workload. To determine the index, it iterates over all candidate indexes, measures their benefit if used along with the winning indexes of earlier iterations. It adds the index with most benefit to the winning set, and iterates till adding an index would violate the space constraint.

Although this algorithm is a very simple, it has been shown to perform better in terms of accuracy than more complex algorithms used in the commercial designers, mainly because of its significantly larger candidate index set [4].

VI. Experimental Results

This section demonstrates the accuracy of PINUM's cost model, and its performance advantage over INUM's model construction. It also shows the benefit of using PINUM on a synthetic star-schema workload.

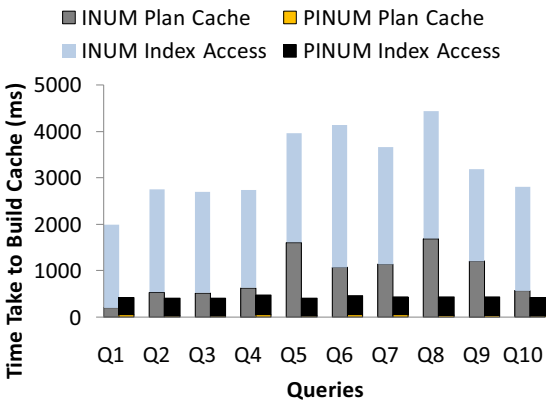

Figure 4 Comparison of cache construction times.

A. Experimental Setup

We implement PINUM on PostgreSQL 8.3.7 on the Windows platform. The implementation in its existing form does not address queries containing complex sub-queries, inheritance, and outer joins. Therefore to investigate the performance and quality of the implementation, we use a synthetic benchmark to study the behaviour of the physical designer in presence of varying query complexity.

The synthetic workload consists of a 10GB star-schema database, with one large fact table, and 28 smaller dimension tables. The dimension tables themselves have other dimension tables and so on. The columns in the tables are numeric and uniformly distributed across all positive integers. We use 10 queries, each joining a subset of tables using foreign keys. Other than the join clauses, they contain randomly generated select columns, where clauses with 1% selectivity, and order-by clauses. In this experiment PINUM generates and searches through 1093 candidate indexes. It identifies 43 useful plans for out of a total of 266 interesting order combinations.

Although the current limitations in our implementation prevent us from using full-blown TPC-H queries, we design the synthetic benchmark to preserve all possible complexity and challenge to our method. The workload consists of a star-schema workload, a well-accepted design for analytical queries. It also favours nested-loop joins more than sort-merge and hash joins. As INUM is less accurate when nested-loop joins are used, our benchmark is more challenging when compared to TPC-H in the context of a cache-based cost model.

B. What-If Index Accuracy

Initially, we use the query optimizer to compute the cost of a query when the indexes are explicitly implemented in the database. Then, we evaluate the cost of the same query by simulating the presence of the same indexes using what-if indexes in the optimizer. We repeat the same experiment 50 times for different set of indexes. The experiment shows that the query cost estimation does not exactly match with the optimizer's cost: the error in the cost estimation was on average 0.33% and the highest observed error was 1.05%. The difference in the estimated cost and the actual cost is a result of the way we compute the number of pages for the indexes. We compute only the sizes of the leaf pages and we do not take into consideration the internal pages of the B-tree index, since they affect the relative page sizes only on very small indexes.

C. Cost Estimation Accuracy

To study the accuracy of PINUM's cost model, we generate 1000 random atomic configurations for each query in the workload. We then compare the cost of the queries using PINUM's cost model and using what-if indexes on the optimizer. Out of ten queries, six had less than 1% error in cost estimation. Further three queries had about 4% error, and only one query had 9% error in cost estimation. This demonstrates PINUM's cache-based cost model provides higher accuracy compared to INUM's cache-based cost model that has 7% error on average [4]. For the single poorly performing query,

PINUM returns the accurate plans. The errors in cost estimation stems from our access cost generation mechanism, as it misses several access paths generated in the join planner. In future, we intend to investigate the addition of these access paths to improve the model's accuracy.

D. Performance Results

Figure 5 demonstrates the efficiency of PINUM while filling the plan cache and collecting the index access costs from the optimizer. The x-axis shows the queries, whereas the y-axis shows the time taken to get the plans for different interesting order combinations and index access costs.

PINUM is typically at least one order of magnitude faster than INUM for cache construction, and 5 times faster for finding the index access costs. PINUM takes a few tens of milliseconds to build the cache for each query, compared to a few seconds required by INUM. Moreover, for queries involving more than three tables in the join clause, PINUM is two orders of magnitude faster than INUM, so PINUM is better suited for complex queries and can scale to a much higher number of queries in the workload. A more intelligent pruning of the unhelpful indexes can speed up the index access cost lookup.

E. Results for the Index Selection Tool

Figure 6 demonstrates the benefit of running the prototypical index selection tool discussed in Section V-E. We run the tool using the 10 queries in the workload, and restrict the tool to suggest indexes taking 5GBs of space on disk. We report the original running times of each of the queries and new running times with the suggested indexes.

Using PINUM's suggested indexes speeds up the workload by 95% on average. PINUM reduces the cost of the most expensive queries by building covering indexes for them. It suggests four covering indexes on the fact table, and three order indexes on the next level dimension tables.

Although we use a synthetic benchmark in the current form of the implementation, the experimental results are indicative of the potential of the index suggestion tool on a real-world workload, and as this benchmark is challenging for INUM, we expect even better performance when using a standard benchmark such as TPC-H.

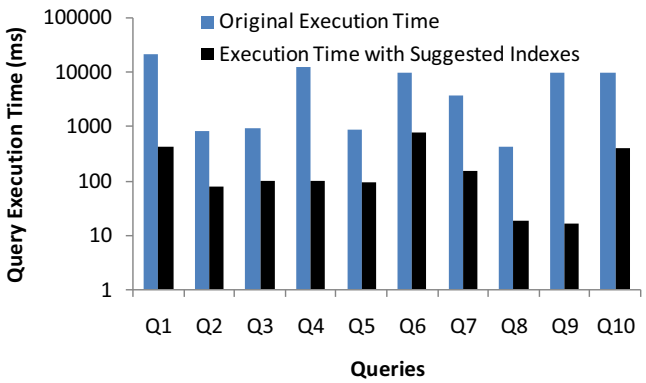

Figure 7 Workload performance improvement by using the index selection tool.

VII. Conclusion and Future Work

In the process of answering what-if queries, the optimizer evaluates many plans that often contain the answer to subsequent questions when evaluating candidate configurations. Harnessing this work instead of throwing it away can speed up candidate evaluation by an order of magnitude, thereby improving solution quality as it reduces the probability that the optimal configuration will be pruned. To demonstrate the effectiveness of the technique we build PINUM, a fast and low-overhead proof of concept cost model for PostgreSQL DBMS. PINUM reduces the construction overhead of a query plan cache by a factor of at least 5, without compromising accuracy. We build a simple index selection tool, and using a large number of candidates select indexes which speed up typical star-schema queries by 95% on average.

Because PINUM removes the overhead barrier from cache-based cost models, in future we intend to use it for building physical designers which suggest partitions, materialized views, and subsequently online workloads.

Acknowledgements

This work was partially supported by Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and SNF funds.

Refrences

[1] Bruno, N. and Chaudhuri, S. 2005. Automatic physical database tuning: a relaxation-based approach. SIGMOD'05.

[2] Zilio, D. C., Rao, J., Lightstone, S., Lohman, G., Storm, A., Garcia-Arellano, C., and Fadden, S. (2004). Db2 design advisor: Integrated automatic physical database design. VLDB'04.

[3] Performance Tuning using the SQLAccess Advisor. http://www.oracle.com/technology/products/bi/db/10g/pdf/twp_general_perf_tuning_using_sqlaccess_advisor_10gr1_1203.pdf

[4] Papadomanolakis, S., Dash, D., and Ailamaki, A. Efficient use of the query optimizer for automated physical design. VLDB'07.

[5] Bruno, N. and R. V. Nehme. Configuration-parametric query optimization for physical design tuning. SIGMOD '08.

[6] Monteiro, J. M., Lifschitz, S. and Brayner, A.: An Architecture for Automated Index Tuning. SBBD, 2006.

[7] Thiem, A. and Sattler, K. An Integrated Approach of Performance Monitoring for Autonomous Tuning. ICDE'09.

[8] Kao, K. and Liao, I. An index selection method without repeated optimizer estimations. *Inf. Sci.* 09.

[9] PostgreSQL documentation manual: http://www.postgresql.org/docs/8.1/interactive/index.html

[10] Chaudhuri, S. and V. R. Narasayya An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. VLDB'97.

[11] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. Access path selection in a relational database management system. SIGMOD '79.