Automated Physical Design: A Combinatorial Optimization Approach

Debabrata Dash

CMU-CS-11-104

February 24, 2011

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee
Anastasia Ailamaki, Chair
Christos Faloutsos
Carlos Guestrin
Guy Lohman, IBM Almaden Research Center

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the National Science Foundation under grant numbers IIS-0429334 and IIS-0431008, the Sloan Foundation under grant number BR4474, the National Aeronautics and Space Administration under grant number NNG06GE23G, and Intel.

Keywords: Database, Self-Managing Database, Physical Design, Combinatorial Optimization, Online Algorithm

Abstract

One of the most challenging tasks for the database administrator is physically designing the database (by selecting design features such as indexes, materialized views, and partitions) to attain optimal performance for a given workload. These features, however, impose storage and maintenance overhead on the database, thus requiring precise selection to balance the performance and the overhead. As the space of the design features is vast, and their interactions hard to quantify, the DBAs spend enormous amount of resources to identify the optimal set of features.

The difficulty of the problem has lead to several physical design tools to automatically decide the design given the data and a representative workload. The state-of-the-art design tools rely on the query optimizer for comparing between physical design alternatives, and search for the optimal set of features. Although it provides an appropriate cost model for physical design, query optimization is a computationally expensive process. Other than affecting the performance of the design tools, the overhead of optimization also limits the physical design tools from searching the space thoroughly – forcing them to prune away the search space to find solutions within a reasonable time. So far it has been impossible to remove query optimization overhead *without sacrificing cost estimation precision*. Inaccuracies in query cost estimation are detrimental to the quality of physical design algorithms, as they increase the chances of "missing" good designs and consequently selecting sub-optimal ones. Precision loss and the resulting reduction in solution quality is particularly undesirable and it is the reason the query optimizer is used in the first place.

In this thesis, we claim that for the physical design problem, the costs returned by the optimizer contain an intuitive mathematical model. By utilizing this model, the physical design problem can be converted to a compact convex optimization problem with integer variables and solved efficiently to attain near-optimal solutions using mature off-the-shelf solvers.

This approach eliminates the tradeoff between query cost estimation accuracy and performance. We invoke the optimizer a small number of times, and then reuse the results of the invocation to create an accurate model. We demonstrate the usefulness of the model by finding near-optimal physical design for workloads containing thousands of queries and thousands of candidate design alternatives. In a more complex online workload scenario, we devise several algorithms with guaranteed competitive bounds for the physical design problem. The proposed online algorithms provide significant speedups while imposing reasonable overhead on the system. This thesis, demonstrates that optimizer–the most complex component of the DBMS–can be modeled in a restricted (yet important) domain. The same approach can be extended to other domains to build accurate and efficient models for the optimization problems, and optimal solutions can be searched in a principled manner.

Contents

1 Automated Physical Design **1**

1.0.1 Challenges in Automated Physical Design 3

1.1 The History of Automated Physical Design 4

1.1.1 Early Days 4

1.1.2 Workload-based Physical Design 6

1.1.3 Physical Design in Commercial Systems 7

1.2 The Need for a new Approach 8

2 Combinatorial Optimization Programs **11**

2.1 Optimization Program 12

2.2 Linear Optimization Programs 12

2.3 Convex Optimization Programs 14

2.4 Combinatorial Optimization Programs 15

2.5 Conclusion 17

3 Efficient Use of the Query Optimizer **19**

3.1 Introduction 19

3.1.1 Accuracy vs. efficiency 20

3.1.2 Our approach 21

3.1.3 Contributions 21

3.2 INUM – The Index Usage Model 23

3.2.1 Setup: The Index Selection Session 23

3.2.2 Reasoning About Optimizer Output 26

3.2.3 INUM Overview 30

3.2.4 Using Cached MHJ Plans . . . 33
3.2.5 Computing the INUM Space . . . 39
3.2.6 Extending the INUM . . . 40
3.2.7 Experimental Setup . . . 46
3.2.8 Experimental Results . . . 49
3.3 Speeding up INUM . . . 54
3.3.1 The PostgreSQL Query Optimizer . . . 55
3.3.2 Harnessing Intermediate Plans . . . 57
3.3.3 Design And Implementation . . . 58
3.3.4 Experimental Results . . . 61
3.4 Applications of INUM's Cost Model . . . 65
3.4.1 Computing Index Interactions . . . 65
3.5 Related Work . . . 69
3.6 Conclusion . . . 71

4 Offline Physical Design **73**
4.1 Introduction . . . 73
4.2 Related Work . . . 76
4.3 Index Selection COP . . . 76
4.3.1 Index Selection for a Query . . . 77
4.3.2 Index Selection for a Workload . . . 78
4.4 Adding Constraints . . . 80
4.5 Adding More Constraints . . . 81
4.5.1 Index Constraints . . . 81
4.5.2 Configuration Constraints . . . 83
4.5.3 Generators . . . 83
4.5.4 Soft Constraints . . . 84
4.5.5 Extending the Constraint Language . . . 85
4.6 Workload as a Sequence . . . 86
4.6.1 Problem Definition . . . 86
4.6.2 The COP for Sequences . . . 87
4.7 Using C-PQO as Cost Model . . . 89

4.8 CoPhy System Design . . . 90
4.8.1 Solving the COP . . . 92
4.8.2 Greedy Algorithm . . . 95
4.9 Experimental Results . . . 97
4.9.1 Experimental Setup . . . 98
4.9.2 Solution Quality Comparison . . . 99
4.9.3 Execution Time Comparison . . . 100
4.9.4 Quality Relaxation vs. Execution Time . . . 102
4.9.5 Comparing Against Commercial Tools . . . 103
4.9.6 More Workloads . . . 107
4.10 Conclusion . . . 111

5 Online Physical Design **113**
5.1 Online Algorithm for Vertical Partitions . . . 114
5.1.1 Related Problems . . . 114
5.1.2 OnlinePD Algorithm . . . 116
5.1.3 Transition Cost Model . . . 119
5.1.4 Query Cost Estimation - QCE . . . 121
5.1.5 Candidate Configurations . . . 127
5.1.6 Architecture . . . 128
5.1.7 Experiments . . . 128
5.2 Online Tuning for Indexes . . . 137
5.2.1 Work Function Algorithm . . . 138
5.2.2 OnlineIT . . . 139
5.2.3 Implementation . . . 140
5.2.4 Cloud Cost Model . . . 141
5.2.5 Experimental Setup . . . 145
5.2.6 Experimental Results . . . 146
5.3 Related Work . . . 150
5.4 Conclusion . . . 151

6 Conclusion **153**

Bibliography **155**

List of Figures

3.1 Database design tool architecture. 20

3.2 Illustration of plan reuse. (a) The optimal plan p_1 for configuration C_1. (b) The cost for C_2 is computed by reusing the cached internal nodes of plan p_1 and adding the costs of the new index access operators, under the assumptions of Section 3.2.1. 25

3.3 (a) Cost estimation with INUM for the example of Section 3.2.1. (b) Complete INUM architecture. 25

3.4 The cost functions for MHJ plans form parallel hyper-surfaces. The unit in the both the axes is the unit used by the optimizer to compare the costs of the plans. The slopes of the surfaces are determined by Eq. 3.1 to be 1. 35

3.5 Modified plan comparison taking into account index I/O costs. The optimal plan for expensive indexes (to the right of the thick line) performs a sequential scan and uses a hash join. 38

3.6 NLJ plan costs for a single table as a function of an index's s_{NL} parameter (System R optimizer). 41

3.7 NLJ plan cost curves for a single table and an unknown cost function of a single index parameter s_I. 43

3.8 Experimental results for TPCH15. (a) Optimizer calls vs. time for an exhaustive candidate set (b) Recommendation quality (c) Optimizer calls vs. time for a heuristic candidate set 47

3.9 Optimizer calls vs. time for the NREF workload 53

3.10 The PostgreSQL optimizer's architecture 55

3.11 The modified query optimizer architecture. The dotted and dashed lines represent the new data flow induced by PINUM's access cost and cache construction optimizations. 60

3.12 Comparison of cache construction times. Note that the time taken to build the PINUM plan cache is close to zero for each query. 63

3.13 Workload performance improvement by using the index selection tool. 64

4.1 Example of the C-PQO plan for the query Q. 89
4.2 Example of the INUM plan for the query Q 89
4.3 Architecture of CoPhy 91
4.4 Results for index selection tools for TPCH15 99
4.5 Execution time of algorithms for varying query sizes. 100
4.6 Execution time of CoPhy at different relaxations from the optimal 101
4.7 Quality comparison against commercial systems 104
4.8 Pareto-optimal curve with the storage constraint and the workload costs as soft constraints 105
4.9 Execution time of CoPhy's solver with varying number of constraints. 106
4.10 Solution quality comparison for SYNTH workloads of increasing complexity. . . . 109
4.11 Tool execution time comparison for SYNTH workloads of increasing complexity. . 110

5.1 Plan graph comparison for configurations with only vertical partition changes. The boxes represent operations, and the numbers next to them show the estimated costs for those operations. The plans remain the same for configurations S_1 and S_2, as they have the same distribution of *filtering* columns, i.e., c and d. Configuration S_3's query plan is significantly different, since it has a different distribution of the *filtering* column. 123
5.2 The structure cached with the key $[query-id, T1((c),(d))]$. Note that all scanning nodes are place holders for the actual partitions and the costs are missing. The plan on right is the estimated plan for the new configuration S_2, with the partitions and estimated costs filled in. 126
5.3 Architecture for online physical design tool. 129
5.4 Affinity matrix (co-access frequency) for ten selected attributes from the *PhotoObjAll* table. 130
5.5 Distribution of response time overhead. 133
5.6 Response time overhead for an adversarial workload. 133
5.7 Efficacy of query plan reuse in cost estimation. 135
5.8 Response time overhead on a daily basis. 136
5.9 The plots for error in estimating the query costs using QCE. The dashed line represents all queries in the workload, while the solid line represents queries with higher than 5 unit cost. The unit of the cost is same as the one used by SQL Server's optimizer to compare the alternative plans. 137
5.10 The Cloud DBMS Architecture 142

5.11 The response times for the original workload, the offline algorithm, and COP algorithm with $w = 100$. 147
5.12 The response times for the workload for various online optimization algorithms. . . 148
5.13 The average cost of running the queries on the system. 149

List of Tables

4.1 The details of the synthetic benchmark. The first row lists the candidate index set's size for the workloads, the second one lists the INUM cache sizes, and the last row shows the average number of tables in the query. 109

Chapter 1

Automated Physical Design

Database management systems are complex software systems containing many tuning knobs. These knobs allow the administrator to achieve optimum performance from the database management system. Over the years, the knobs have become very complex, causing the administrator to be overwhelmed by the number of possibilities and their implications. From recent surveys of a DBA group, typical business spends about 60% of its resources to designing and managing the database, compared to about 13% of acquiring it. Training and installation account for the remaining expenses [32].

The database design process usually comprises the following steps[45]:

1. *Requirement Analysis* to determine the data, the relationship between the data elements, and the queries required by the software on top of the data.

2. *Logical Design* to determine the conceptual model of the database from the user requirements and the database tables.

3. *Physical Design* to ensure that the queries on the database are executed efficiently.

4. *Monitoring and Refinement* is an iterative step, where the database is monitored for changes

in user requirements, or data characteristics. Once a change is detected, the logical and physical design steps may be repeated. Typically the physical design is repeated more often than the logical design, as the workload tends to change faster than the data characteristics.

For the logical design, the administrator finds the relationships between the data elements and builds the relations, their attributes and their dependencies. This process requires thorough understanding of the data. Tools like ER diagrams help the administrator encapsulate and understand the details of such relationship and then materialize them into the relational databases. Once the dependencies are determined, determining the right schema is straightforward using one of the available automated tools[72].

The physical design of the database is the physical organization of the data on the disk, which allows the queries to run faster compared to the basic relational tables. The database administrator (DBA) typically builds indexes, materialized views, and partitions to help the query performance. Hence forth, we will denote such structures as design features. Although these features help improve the query performance, they are numerous. For example a query using n columns from a table can possibly use $2^n n!$ possible indexes. Moreover, these indexes interact with each other, since some query plans can be feasible only when two indexes are present in the system, and not possible otherwise. Therefore, physically designing the system is expensive in both time and money.

Automatic physical design tools alleviate this problem by helping the administrator achieve the goals she desires from the system, by configuring the system automatically. They search through the large set of design features, taking into account their interactions and suggesting features which boost the performance of queries. While searching through the design space, they also consider the constraints set by the DBA. To define the problem more formally:

Automated Physical Design Problem: *Given a workload of queries W, and a set of constraints C, the physical designer should suggest design features to achieve goals G.*

One common instance of such problem is: Given a workload containing a set of queries, the physical designer should reduce the workload cost as much as possible, while keeping the size of the design features less than a given constant M.

1.0.1 Challenges in Automated Physical Design

We list the main challenges in the automatic physical below.

Huge search space: The difficulty in automatic physical design lies in the huge search space possible for the design features. As discussed earlier, the number of possible features is exponential in the number of columns used in the workload, and the designer has to consider the possible interaction between the design features to select them in a group – further increasing the search space.

Workload selection: Furthermore, there is no standard way to collect the workloads. Typically the DBAs collect the heavy hitters from the SQL log that run on the database. The workload, however, evolves over time. And the optimized design for the heavy hitters becomes irrelevant in the future heavy hitters as they may require different design features due to the usages of their columns. Also, the workload often contains workflows, instead of a sequence of queries. The workflow allows the DBA to drop certain design features after running query and later rebuild them again.

The workload where there is no order on the queries is considered a *set-based workload*. The workload where the order of the queries is known is known as a *sequence-based workload*. Finally, sometimes the workload is completely unknown. In this case the user of the database system creates ad-hoc queries and runs them on the DBMS. This type of workload is called an *online workload*.

Data update: The underlying data can change. When the data changes, the indexes and ma-

terialized views need to be updated with the new data. The bigger problem is that a data change can also make some indexes obsolete. For example, indexes are usually beneficial when the query condition using the index is very selective. If the data update adds lots of rows for a condition and makes it less selective, then the index becomes less useful. It may be possible to use a new index or discard the index altogether to build other physical design structures. This update of the selectivity requires that the design selection is either agnostic to the update or adapt to the update efficiently.

All the factors discussed above make the physical design of databases challenging not only for the DBA, but also for an automated design selection mechanism.

1.1 The History of Automated Physical Design

1.1.1 Early Days

In the early days of DBMS research, Yum et al. proposed selecting indices to speedup workload [46]. The authors recognized the importance of selecting indexes, as they are generic physical design structures. They built a mathematical model of the index usage and the index update overheads. Using empirical statistics to build this model, they associated an indicator function on the following parameters: index size, frequency of update, and insertion of records, frequency of reference to the indexed column in queries, frequency of a full table scan if the column is not indexed. When the indicator function is positive, the index is built and dropped otherwise. Hammer et al. extended this mechanism by refining the statistics collection and workload characterization process [61].

A related but more theoretical approach was proposed in references [42, 50, 60, 69]. In these approaches, the workload was modeled as a collection of attributes, along with their access and update probabilities. Then a model is built to estimate the cost of the entire workload. They built a

cost model using these probabilities to identify useful indexes. Finally, they selected the secondary indices using an optimization algorithm, such as greedy selection. Comer analyzed the problem of index selection and proved that the problem is NP-complete by reducing the satisfiability problem into the index selection problem [23]. Note that the secondary index selection problem focused only on selecting single-column indices. Since the natural approach to address an NP-complete problem is to develop approximation algorithms, several researchers proposed approximation algorithms to solve the problem efficiently, but with a known distance from the optimal set of indices [6, 40]. Another way to address the complexity is to reduce the problem space by decomposing the larger problem into an independent set of sub-problems. Several researchers tried to identify the composability of the index selection problem by using several models for the workload characteristics [7, 74]. These models try to divide the database into smaller set of relations, and find the suitable indexes for those small sets iteratively. For example, Whang et al. [74] consider a single relation at a time and isolate the effect of join between relations by using a "coupling factor". The coupling factor was accurate for the nested-loop joins mechanisms, but fails to take into account the modern join mechanisms.

In parallel, researchers also discovered that the data organization improves the database performance. Cardenas et al. proposed a cost model that uses the disk characteristics and its impact on the data organization together to find beneficial data organizations [16]. Similarly, Babad and Hoffer et al. proposed partitioning the data files into subcomponents, appearing to be the first attempt towards doing horizontal and vertical partitioning in file systems [5] [36]. The results were demonstrated to be useful in the files stored in the database systems as well. Babad developed a non-convex optimization problem using a cost model for the partitions, and then solved it to find the partitions. Solving non-linear and non-convex optimization methods to solve the physical design problem incurs a very high overhead and does not scale to workloads containing hundreds of queries. Hoffer et al. used the clustering of the attributes in the workload instead. They first create an attribute usage matrix, in which the rows represented the queries and the columns repre-

sented the columns in the database. They then build an "column affinity matrix" to determine the columns which occur together. They clustered such matrix to store the set of columns which have high "affinity" and built vertical partitions from them. This mechanism is still useful if purely vertical partitions are required, but does not take into account the interactions between the partitions and other physical design structures such as indexes.

1.1.2 Workload-based Physical Design

In 1988 Finkelstein et al. published a paper regarding the design of System R's database design tool called DBDESGN [28]. The tool was later commercialized under the name of Relational Design Tool (RDT) by IBM. In this paper, the authors made the following important contributions:

- The complexity of the queries by that time had made the earlier probabilistic counting methods irrelevant. Therefore, the cost had to be determined per query, rather than per column in the workload.
- They also recognized that the optimizer should be used to estimate the cost of the indices.
- So far, all work on the physical design focused on selecting secondary indices, assuming that the primary key or the clustered index has been determined by the logical design process. In this paper, Finkelstein et al. propose to generalize all the access paths under the physical design process.
- Finally, they built indices for multiple tables. Earlier attempts tried to suggest indexes for queries containing multiple tables using the separability of the indices.

Dabrowski et al. proposed yet another practical approach by integrating developments in knowledge-based decision systems to the physical design process [26]. Rozen et al. extended the DBDESGN work to propose a general framework that divides the physical design into two

steps. In the first step, the optimal indices for each query is determined, and then in the second stage the indices are "combined" into multicolumn indexes to produce a good physical design [56].

Cheonni et al. proposed to combine the idea of Finkelstein et al. to generate good candidate indices using the information achieved by the optimizer, and the idea of Dabrowski et al. that knowledge rules can narrow the search space in finding good indices [21].

1.1.3 Physical Design in Commercial Systems

In the decade after the paper by Finkelstein et al., optimizers had developed significantly. They supported multitudes of physical design structures, such as indexes, multi-column indexes, materialized views, and partitions. Optimizers contained many rules apart from the System R's cost models to select these physical design features. Therefore, earlier approach of using System R model, used by Finkelstein et al. was even more important and adopted by all commercial systems.

Chaudhury et al. developed an API which called what-if API into the optimizer [19]. Using these APIs the physical designer can directly invoke the optimizer on "simulated" indices and other physical design structures. The cost reported by the optimizer is more accurate than the cost models used in earlier approaches, as the optimizer implementation had drifted from those simplistic models significantly. This API became very popular among the researchers, and all research afterwards used the what-if optimizers. This paper started the commercial era of the physical design tools. Chaudhury et al. describe all the advances made possible because of this API in the survey paper Ref. [17]. What follows is a very brief summary of the research discussed in the survey paper. The paper also proposed techniques for finding candidate "configurations" or combinations of indexes, and ways of finding the cost of those configurations by using optimizer caches.

Microsoft followed up on the paper to build a physical design tool in both SQL Server 2000 [58], and SQL Server 2005 [2]. The latter version of the physical design tool integrates selection of indexes, materialized views, and partitions together.

DB2 also shipped a physical design tool in 1999, called the Design Advisor [68]. This work improved on the earlier technique by utilizing the optimizer itself to suggest the candidate indexes. It also used a variant of the knapsack algorithm to find the indexes. Later, the tool integrated selection of materialized views, clustered indexes, and partitions [75].

Oracle shipped a version of their physical designer tool in Oracle 10g [27], which processes the workload traced by the system and then suggests indexes and materialized views.

Researchers have also tried to model the workload in a different manner, such as, sequences [4], and as an online sequence of queries [13, 59, 63]. New types of physical design structures have also been developed, requiring new cost models and physical design tuning [39, 41, 57]. Several researchers also proposed a non-traditional approach such as genetic algorithm to solve the complexity of physical design [44]. Bruno et al. [14] extended the usefulness of the physical design process by incorporating multitudes of constraints that the DBAs need to specify in a real-world physical design scenario.

1.2 The Need for a new Approach

While existing commercial approaches solved the issue of divergence of the design tool's cost model from the optimizer – calling the optimizer as a black box can be expensive. For each call to the optimizer, it needs to recreate the entire optimization process, starting from determining the selectivity, selecting the appropriate design features to optimize, and then selecting the optimal join order and methods. This process takes significant amounts of time, and from our measurements modern physical designers spend about 90% of their time calling the optimizer.

This not only reduces the efficiency of the physical designer, but also the quality of the solution proposed. Since most of the algorithms focus on reducing the number of candidates and querying the optimizer only when necessary, they need to greedily prune away the search space. At every greedy pruning step, the search algorithm possibly sacrifices some quality in the solution.

Moreover, the greedy algorithms do not provide any feedback regarding the distance from the optimal solution. So, the administrator does not have any idea as to how good the suggested solution is, and if she should give the designer more time to achieve a better solution. Estimating the distance from the optimal is critical in the presence of complex constraints. Such constraints could make the final solution infeasible, but the greedy algorithm cannot identify the infeasibility and so can keep on searching for a long time before giving up.

There is an alternative approach that ameliorates this situation significantly. In the area of Operations Research, more and more problems are being posed as convex optimization problems and then solved efficiently using mature solving techniques. The solvers are general and can handle thousands to millions of constraints and solve them in minutes. They also provide a running estimate of the distance of the current solution from the real optimal. This helps the DBA detect infeasible problems and also allows her to estimate the amount of time it would take to reach the optimal solution.

Converting a physical design problem into a convex optimization problem is not an easy task. Since the optimizer is complex, and the search space is huge. In this thesis, we present two techniques that allow the application of the convex optimization to the physical design problem.

More formally, in this thesis we claim that:

1. *In the context of physical design, the costs of queries contain an intuitive mathematical model.*

2. *This model can be exploited to search through the candidate design features in a principled*

manner – without heuristic pruning – to improve both the quality of the suggested features and the efficiency of their selection.

We support this claim by discussing the cost model for the optimizer for the physical design problem in Chapter 3. Chapter 4 uses this model to build a integer linear program to solve the physical design problem in which the workload is predetermined. Finally, Chapter 5 solves the physical design problem in the presence of online and evolving workloads. Before diving into the details of our approach, Chapter 2 provides an introduction to the mathematical optimization, as we use this optimization technique extensively in the reminder of the thesis. Finally, we conclude in Chapter 6.

Chapter 2

Combinatorial Optimization Programs

Recently, the optimization programs have become a tool for many disciplines of computer science, such as networking, machine learning, VLSI design. The efficiency of the tools solving the optimization programs allow large scale problems to be solved reliably and efficiently. This thesis demonstrates that even the most complex optimization problems in the database optimization can be reduced to the combinatorial optimization programs and the existing solution mechanisms can be leveraged to find the solutions scalably and efficiently.

In this chapter, we describe a very high level introduction to the optimization programs, and describe the techniques used to solve them. This chapter provides background knowledge for the combinatorial programs for database physical design in the rest of the thesis. This chapter is strongly influenced by Steven Boyd's excellent book titled "Convex Optimization" [10]. We highly recommend the book for more detailed analysis of the programs and their solutions.

2.1 Optimization Program

An optimization program is defined by an objective $f_0(x)$ on a vector of variables $x = (x_1, x_2, \ldots, x_n)$ where $x_i \in \mathbb{R}$ or $x_i \in \mathbb{N}$. The program also contains a set of constraints the x_i variables must satisfy: $f_j(x), j \in [1, m]$. The full form of the optimization program can be written as:

$$\begin{aligned} &\text{minimize } f_0(x) \\ &such that\, f_j(x) \geq b_j \forall j = 1 \ldots m \end{aligned}$$

Note that, other forms of optimization can be converted to this form using minor variations. For example, if the objective is to maximize a function $f_0'(x)$ then we minimize $-f_0'(x)$. In general, without any knowledge about the functions f_j, it is hard to solve such optimization program efficiently. The solving process can be much more efficient, if the behavior of the functions contain some properties. We now discuss these properties, and mechanisms to solve the optimization problem using those properties.

2.2 Linear Optimization Programs

For an optimization program to be called a *linear program*, the function f_j must be linear functions of the x_i variables. More formally, a function f is called a linear function if:

$$f_j(\alpha x + \beta y) = \alpha f_j(x) + \beta f_j(y)$$

Linear programs are common in the real world engineering problems, and solving them efficiently has allowed rapid adoption of the linear program solvers in many disciplines.

Let's consider a trivial example of a linear program to get a feel about it. Assume for simplicity that there are only two food items A and B. Further, assume that there are only two products people

need to stay alive, p and q. Each day a person must consume at least 60 units of p and at least 70 units of q to stay alive. Let us assume that one unit of A costs \$20 and contains 30 unit of p and 5 units of q, and that one unit of B costs \$2 and contains 15 units of p and 10 units of q. The goal is to find the cheapest diet which will satisfy the minimum daily requirements.

We assign variables $x = (x_1, x_2)$ to the amount of food items consumed per day, then we need to minimize the total cost:

$$f_0(x) = 20x_1 + 2x_2$$

But we also have to make sure that the minimum requirement for the day is satisfied as well using the following constraints:

$$\begin{aligned} f_1(x) = & \quad 30x_1 + 15x_2 \geq 60 \\ f_2(x) = & \quad 5x_1 + 10x_2 \geq 70 \\ f_3(x) = & \quad x_1 \geq 0 \\ f_4(x) = & \quad x_2 \geq 0 \end{aligned}$$

The functions $f_0, ..., f_4$ define a linear program, that when solved for the values of x_1 and x_2 provides the amount of A and B required to satisfy the daily nutrition requirements, while incurring the minimum cost.

The most popular way to solve the linear programs is Danzig's Simplex method [**?**]. The Simplex algorithm models the optimization problem in geometric terms. Each constraint specifies a half-space in n-dimensional Euclidean space, and their intersection is a polytope – a n-dimensional generalization of the two-dimensional polygons. To minimize the objective, the algorithm needs to

search for the minimum point $f_0(x)$ which intersects this polytope. For linear programs $f_0(x)$ can be written as $c^T x$, therefore the objective function can be viewed as a hyperplane in the direction of the vector c. Intuitively, the last hyperplane to intersect the feasible region will either just graze a vertex of the polytope, or a whole edge or face. In the latter two cases, it is still the case that the endpoints of the edge or face will achieve the optimum value. Thus, the optimum value will always be achieved on (at least) one of the vertices of the polytope. The simplex algorithm applies this insight by walking along edges of the (possibly unbounded) polytope to vertices with lower objective function value, and terminates when it either reaches the minimum value, or it reaches the unbounded face of the polytope (signifying that there is no optimal solution).

The simplex method's worst-case complexity is exponential, it is very efficient in practice, generally taking $2m$ to $3m$ iterations at most. More general techniques such as interior-point methods solve the linear programs in polynomial time in the worst-case.

2.3 Convex Optimization Programs

A more generic form of optimization program is called "convex optimization programs" if the f_j functions are "convex" in nature. A function is said to be convex if it satisfies the following constraint:

$$f_j(\alpha x + \beta y) \leq \alpha f_j(x) + \beta f_j(y)$$

Where $\alpha, \beta \in [0,1]$ and $\alpha + \beta = 1$.

This is more general than the linear program, as the equality is replaced by the inequality, and the functions can be non-linear in nature. The non-linearity of the functions allow a wider range of real-world problems to the mapped to the convex optimization programs.

The convex optimization programs can be solved using interior-point methods. Describing the

interior point method is beyond the scope of this chapter, but intuitively, as Simplex traversed along the edge of the constrained area, in the interior point methods the algorithm traverses inside the convex region to arrive at the solution. As we mention earlier, even though the interior methods are not as efficient as the simple method in the average case, in the worst-case they can be solved in $O(max(n^3, n^2m, F))$, where F is the cost to find the first and second derivatives for the objective function.

2.4 Combinatorial Optimization Programs

In this type of programs, we add integrality constraint to the convex optimization programs. In other words, we consider a subset z of the variables x, such that $z_k \in N$. Adding the integrality condition makes the problem of solving the combinatorial optimization programs NP-hard. In practice, there exist many heuristics to solve the the combinatorial optimization problems efficiently. These heuristics provide solutions close to the optimal in seconds for the real-world problems containing thousands of variables and constraints. The algorithms arrive at the solution by using three fundamental techniques: Branch-bound, Branch-cut, and Parallel solving.

Branch-bound: A branch-and-bound technique requires two sub-techniques. The first one is a splitting procedure that, given a set S of candidates, returns two or more smaller sets $\{S_1, S_2, \ldots, S_m\}$, whose union covers S. Note that the minimum of $f(x)$ over S is $minV_i$, where each V_i is the minimum of $f(x)$ within S_i. This step is called branching, since its recursive application defines a tree structure (the search tree) whose nodes are the subsets of S.

The bounding technique computes upper and lower bounds for the minimum value of $f(x)$ within a given subset S. Typically the lower bound is achieved by using LP-relaxation technique. In LP-relaxation the integral constraints are removed to allow $x \in R$. The selection of the upper bound is typically application dependent.

The key idea of the branch-and-bound algorithm is: if the lower bound for some tree node (set of candidates) S_i is greater than the upper bound for some other node S_j, then S_i may be safely discarded from the search. This step is called pruning, and is usually implemented by maintaining a global variable m (shared among all nodes of the tree) that records the minimum upper bound seen among all sub-regions examined so far. Any node whose lower bound is greater than m can be discarded.

The recursion stops when the current candidate set S is reduced to a single element; or also when the upper bound for set S matches the lower bound. Either way, any element of S will be a minimum of the function within S.

Branch-cut: For branch and cut, the lower bound is again provided by the linear-programming (LP) relaxation of the integer program. If the optimal solution to the LP is not integral, this algorithm searches for a constraint which is violated by this solution, but is not violated by any optimal integer solutions. This constraint is called a cutting plane. When this constraint is added to the LP, the old optimal solution is no longer valid, and so the new optimal will be different, potentially providing a better lower bound. Cutting planes are iteratively until either an integral solution is found or it becomes impossible or too expensive to find another cutting plane. In the latter case, a traditional branch operation is performed and the search for cutting planes continues on the subproblems. This approach was pioneered by Gomory [31].

Parallel solving: Since the branch-bound process can be easily parallelizable, modern solvers provide different branches of the tree to different processors, and then solve the problem in parallel. In modern multi-processing architectures this method solves problems faster compared to using one processor.

2.5 Conclusion

In this section we provide a high level overview of the mathematical optimization programs, and their variations. We also discuss the algorithms used to solve the optimization programs, and their complexities. We use this background knowledge to build efficient optimization programs for the physical design problem and solve them efficiently using these well known techniques.

Chapter 3

Efficient Use of the Query Optimizer

3.1 Introduction

As database applications become more sophisticated and human time becomes increasingly expensive, algorithms for automated design and performance tuning for databases are rapidly gaining importance. A database design tool must select a set of design objects (e.g. indexes, materialized views, table partitions) that minimizes the execution time for an input workload while satisfying constraints on parameters such as available storage or update performance. Design tasks typically translate to difficult optimization problems for which no efficient exact algorithms exist [20] and therefore current state-of-the-art design tools employ heuristics to search the design space. Such heuristics trade accuracy for performance by pruning possible designs early without spending time evaluating them. Recent trends, however, dictate that application schemas are becoming richer and workloads are becoming larger – which increases the danger of compromising too much accuracy for the sake of performance. Automated data management cannot rely entirely on aggressive pruning techniques anymore to remain efficient; we need a way to efficiently evaluate large portions of the design space without compromising accuracy and with acceptable performance.

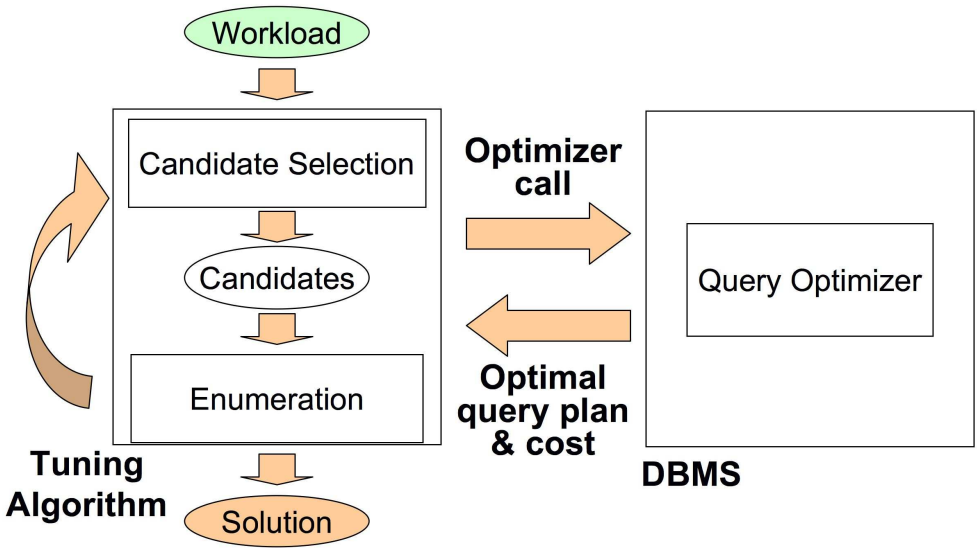


Figure 3.1: Database design tool architecture.

3.1.1 Accuracy vs. efficiency

Figure 3.1 outlines the two-stage approach typically employed by database design tools [1, 19, 33, 75]. The *candidate selection* stage has the task of identifying a small subset of "promising" objects, the *Candidates*, that are expected to provide the highest performance improvement. The *enumeration* stage processes the candidates and selects a subset that optimizes workload performance while satisfying given resource constraints.

Although tools differ in how they implement Figure 3.1's architecture, they all critically rely on the query optimizer for comparing different candidates, because it provides accurate estimates of query execution times. The downside of relying on the optimizer is that query optimization is extremely time-consuming. State-of-the-art tools spend most of their time optimizing queries instead of evaluating as many of the "promising" candidates or candidate subsets as possible. Quoting from a recent study [11]: "...we would require hundreds of optimizer's calls per iteration, which becomes prohibitively expensive". Indeed, our experiments show that on average 90% of the running time of an index selection algorithm is spent in the query optimizer.

3.1.2 Our approach

This chapter presents the INdex Usage Model (INUM), a novel technique for deriving query cost estimates that reconciles the seemingly contradictory goals of high performance and estimation accuracy. INUM is based on the intuition that, although design tools consider an immense space of different alternative designs, the number of different optimal query execution plans, and therefore the number of different possible optimizer outputs, is much lower. Thus, it makes sense to *cache and reuse* the set of plans output by the optimizer, instead of performing multiple invocations only to compute the same plan multiple times.

Like the optimizer-based approaches, INUM takes as input a query and a physical design (a set of indexes) and produces as output an estimate for the query cost under the input physical design. Unlike the optimizer-based approaches, however, INUM returns the same value that would have been returned by an equivalent optimizer invocation, without actually performing that invocation. As a consequence of drastically decreasing the dominant overhead of query optimization, INUM allows automated design tools to execute 1.3 to 4 times faster *without sacrificing precision*.

3.1.3 Contributions

This chapter's contribution to automated index selection algorithms is as follows:

1. Faster index selection. Our experiments demonstrate that an index selection algorithm with INUM provides three orders of magnitude faster cost estimation, after the initial calls to the optimizer. When factoring in the precomputation phase that involves the optimizer, we measured execution time improvements of 1.3x to 4x – without implementing any of the techniques proposed in the literature for optimizing the number of cost estimation calls, which would result in even higher speedup for INUM. a wider design space than an optimizer-based tool. From a different point of view, faster index selection by an INUM-enabled tool

translates into the capability to evaluate a wider design space (solve a bigger problem) for a given evaluation time than the traditional optimizer-based tool.

2. Improved accuracy. INUM allows existing search algorithms (such as greedy search [1, 19]) to examine three orders of magnitude more candidates. Evaluating more candidates benefits solution quality because it reduces the number of "promising" candidates that are overlooked as a result of pruning. According to our experiments, INUM evaluates a candidate set of more than a hundred thousand indexes for a TPC-H based workload, performing the equivalent of millions of optimizer invocations within four hours – a prohibitively expensive task for existing optimizer-based tools. The solution derived from this "power" test improves the solution given by a commercial tool by 20%-30% for "constrained" problem instances with limited storage available for indexes.

3. 100% compatibility with existing index selection tools. INUM can be directly integrated into existing tools and database systems, because it simply provides a cost estimation interface without any further assumptions about the algorithm used by the tool.

4. Improved flexibility and performance for *new* index selection algorithms. Recent work on index selection improves solution quality by dynamically generating candidates, based on a set of transformations and partial enumeration results. The *relaxation-based* search in [11] generates thousands of new candidates combinations, thereby making optimizer evaluation prohibitively expensive. INUM could help by replacing the approximation logic currently used in Ref. [11], allowing the algorithm to use exact query costs (as opposed to upper bounds) thereby avoiding estimation errors and the corresponding quality degradation. We propose a novel database design approach, closely integrated with INUM, in the next chapter.

5. Even faster index selection by exploiting optimizer structures. We extend INUM by using the internal optimization structure in PostgreSQL to build a much faster cost estimation mech-

anism. This extension removes the overhead of INUM's cache construction, and enables efficient online index selection.

In this chapter we first discuss INUM in Section 3.2, then discuss the integration of INUM with PostgreSQL in Section 3.3. Although index selection is the original motive for INUM, we discuss how to use INUM for a different scenario – to determine the index interactions – in Section 3.4. Finally we discuss the related work and conclude.

3.2 INUM – The Index Usage Model

We show, through an example, how the INUM accurately computes query costs while at the same time eliminating all (but one) optimizer calls. For this section only, we make certain restrictive assumptions on the indexes input to INUM. This section sets the stage for the complete description of INUM in the next sections.

3.2.1 Setup: The Index Selection Session

Consider an index selection session with a tool having the architecture of Figure 3.1. The tool takes as input a query workload W and a storage constraint, and produces an appropriate set of indexes. We will look at the session from the perspective of a single query Q in W. For the sake of simplicity of explanation, let Q be a select-project-join query accessing 3 tables (T_1, T_2, T_3). Each table has a join column ID, on which it is joined with the other tables. In addition, each table has a set of 4 attributes ($a_{T_1}, b_{T_1}, c_{T_1}, d_{T_1}$, etc.) on which Q has numerical predicates of the form $x \leq a_{T_i} \leq y$.

The automated design tool generates calls to the optimizer requesting the evaluation of Q with respect to some index *configuration* C. For the remainder of this chapter we use the term "config-

uration" to denote a set of indexes, according to the terminology in previous studies [19].

To facilitate the example presented in this section, we assume that the configurations submitted by the tool to the optimizer *contain only non-join columns*. No index in any configuration contains any of the *ID* columns and the database does not contain clustered indexes on *ID* columns. This is an artificial restriction used to facilitate the example in this section *only*.

Our techniques naturally extend to index unions or intersections, however we maintain the single access method per table throughout the chapter, for simplicity. We assume that queries use at most one index per table. For this example, the configurations submitted on behalf of query Q are represented as tuples (T_1: I_{T_1}, T_2: I_{T_2}, T_3: I_{T_3}), where any of the I_{T_i} variables can be empty. [1].

The optimizer returns the optimal query execution plan for Q, the indexes in C utilized by the plan and the estimated cost, along with costs and statistics for all the intermediate plan operators (Figure 3.2 (a) shows an example optimal query plan). There will be multiple optimizer calls for query Q, one per configuration examined by the tool. Ideally we would like to examine a large number of configurations, to make sure that no "important" indexes are overlooked. However, optimizer latencies in the order of hundreds of milliseconds make evaluating large numbers of configurations prohibitively expensive.

Existing tools employ pruning heuristics or approximations (deriving upper bounds for the query cost [11]) to reduce the number of optimizer evaluations. In the next sections we show how to obtain accurate query cost estimates efficiently, without any query optimization overhead.

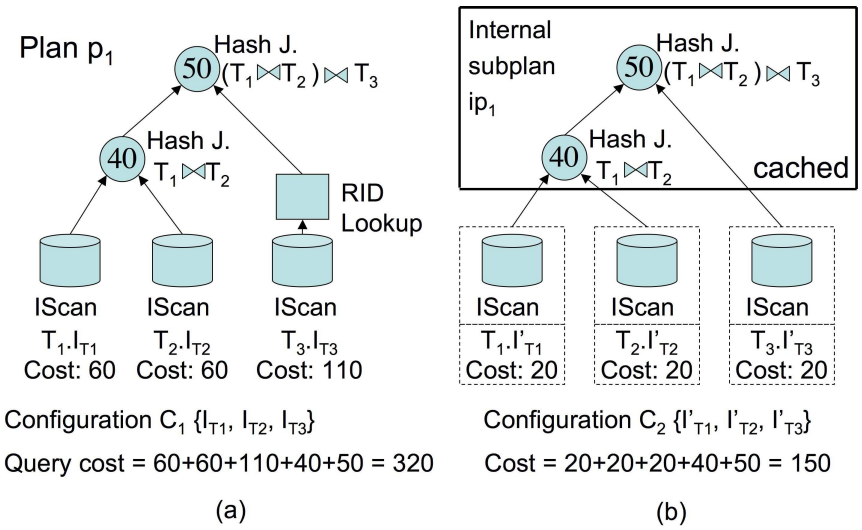


Figure 3.2: Illustration of plan reuse. (a) The optimal plan p_1 for configuration C_1. (b) The cost for C_2 is computed by reusing the cached internal nodes of plan p_1 and adding the costs of the new index access operators, under the assumptions of Section 3.2.1.

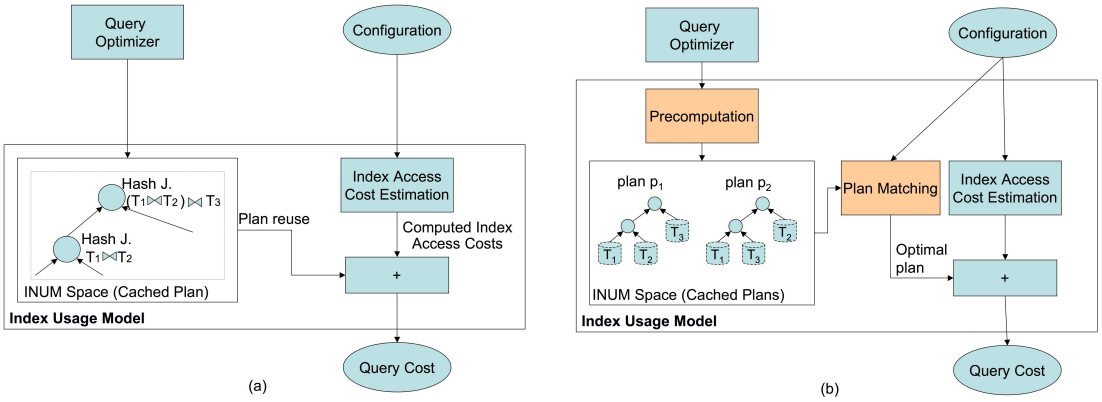


Figure 3.3: (a) Cost estimation with INUM for the example of Section 3.2.1. (b) Complete INUM architecture.

3.2.2 Reasoning About Optimizer Output

Assume we have already performed a single optimizer call for the SPJ query Q and a configuration C_1 and obtained an optimal plan p_1. We first specify the procedure for *reusing* the information in plan p_1, in order to compute Q's cost for another configuration C_2. The optimality of plan p_1 with respect to configuration C_2 is discussed later.

To compute the cost of Q under a new configuration C_2, we first compute the *internal subplan* ip_1 from p_1. The *internal subplan* is the part of the plan that remains after subtracting all the operators relevant to data access (table scans, index scans, index seeks and RID lookups). The internal structure of the plan (e.g. join order and join operators, sort and aggregation operators) remains unchanged.

Next, for every table T_i, we construct the appropriate data access operator (index scan or seek with RID lookup) for the corresponding index in C_2 (or a table scan operator if the index does not exist). The data access operators are appended to ip_1 in the appropriate position.

Finally, Q's cost under C_2 can be computed by adding the total cost for ip_1 (which is provided by the optimizer) to the cost of the new data access operators corresponding to the indexes in C_2.

Given p_1 and a new configuration C_2, replacing the optimizer call by the above reuse procedure allows for dramatically faster query cost estimation. Since we have already computed plan p_1 (optimal for C_1), most of the work is already done. The only additional cost is computing the costs of the data access operators in the third step of the reuse procedure: this can be done efficiently and precisely by invoking only the relevant optimizer cost models, without necessitating a full-blown optimization. The reuse procedure is more efficient than an optimizer call because it avoids the overhead of determining a new optimal plan. Figure 3.2 (b) shows how plan p_1 is reused with a new configuration C_2 and the new query cost.

[1]Our technique only needs to characterize each index by its access cost and the ordering it provides. Both properties can be computed for sets of indexes combined through a union or intersection operator

We define two desirable properties of the above reuse process: "*correctness*" and "*accuracy*". The plan returned by the cache-reuse mechanism is "correct", if the plan can be executed by the DBMS query execution engine. For example, the plan cannot use an index scan for a predicate, unless the predicate column is also in the index. Similarly, the plan should not use merge-join, unless an index provides the order, or a descendant of the merge-join node first sorts to provide the order. A plan is called "accurate", if the total cost of the plan matches the optimal plan returned by the optimizer. Since it is not always possible to identify fully accurate plans, we use the relative difference between the costs of the optimal plan returned by the optimizer and the reused plan to measure the accuracy.

Since our goal is to maintain accuracy, we must also consider the *correctness* of reusing a plan p with some input configuration. the correctness ensures that the plan returned from the cache is exactly as it would have been returned by the optimizer. For our example, reusing plan p_1 for C_2 will yield erroneous results if the subplan ip_1 is not optimal for C_2. In other words, before we reuse plan p for some configuration c we must be in a position to prove that p is optimal for c and this without invoking the optimizer! Focusing on correctness is the differentiating factor between our approach and existing techniques based on "local plan modifications" [11, 12]. The latter do not consider plan optimality and therefore only guarantee the computation of an upper bound for query costs.

We now present a correctness proof, based on simple reasoning about optimizer operation. Specifically, for the scenario of this section, we prove that there exists a single optimal subplan for Q, regardless of the configuration C_2, as long as the non-join column constraint of Section 3.2.1 is satisfied. Since p_1 is computed by the optimizer, it has to be the optimal plan and thus it can be safely reused according to our reuse procedure.

We intuitively justify this argument by considering how the indexes in C_2 affect the query plan cost: Without the join column, there is no reason for C_2 to "favor" a particular join order or join

algorithm, other than those in p_1. Since the indexes in C_2 are not more "powerful" than those in C_1, there is no reason for p_1 to stop being optimal. Notice that the non-join column restriction is critical: if it is violated, the reuse procedure will yield incorrect results.

We formalize using the following theorem:

Theorem 3.2.1 *For a query Q, there is a single optimal plan for all the configurations that do not contain indexes on Q's join attributes.*

Proof We prove Theorem 3.2.1 by contradiction. Let plans p_1 and p_2 be optimal plans for configurations C_1 and C_2 respectively and let p_1, p_2 differ in their internal nodes (different join order, for instance). Let ip_1 and ip_2 be the internal subplans of p_1, p_2 and c_{ip_1}, c_{ip_2} their total costs, with $c_{ip_1} < c_{ip_2}$.

We first show that the cost of accessing the indexes in C_1 and C_2 is independent of the internal structure of the plan chosen. Since the join attributes are not indexed, any operator in ip_1 and ip_2 will access its corresponding index (or table) with an optional RID lookup. The cost of the scan depends only on the columns of the index and the selectivities of relevant query predicates and is the same regardless of the plan. Thus the index access costs for the indexes in C_1 and C_2 are the same for plans p_1 and p_2.

Next, we show that the internal subplans ip_1 and ip_2 can be used with the indexes of both C_1 and C_2 (according to the reuse procedure) and that their costs will be the same: Since we assume no join columns, there is no reason why ip_1 cannot use the indexes in C_2 and vice-versa. In addition, since C_1, C_2 do not involve join orders, order-by, group-by, or differed materialization of the columns, the only other way a data access operator can affect the internal subplan is through the size and the cardinality of its output, which is the same regardless of the access method used.

Thus ip_1 and ip_2 can use the indexes in C_1 and C_2 interchangeably and the index access costs and internal plan costs remain the same. Since $c_{ip_1} < c_{ip_2}$ and the index access costs are the same,

using ip_1 for C_2 is cheaper than using ip_2, and thus p_2 is not the optimal plan for C_2, a contradiction.

Theorem 3.2.1 means that only a single call is sufficient to efficiently estimate $Q's$ cost for any configuration, under the no-join column restriction. Our result can be generalized using the notion of an interesting order:

Definition 3.2.1 *An interesting order is a tuple ordering specified by the columns in a query's join, group-by or order-by clause [66].*

Definition 3.2.2 *An index covers an interesting order if it is sorted according to that interesting order. A configuration covers an interesting order if it contains an index that covers that interesting order.*

Although we used a select-project-join query to derive Theorem 3.2.1, the same reasoning could be applied to queries involving group-by, or order-by clauses. For a query with joins, group-by or order-by clauses, only a single plan (and a single optimizer call!) is sufficient to estimate its cost, for all the configurations that do not cover the query's interesting orders.

Figure 3.3 (a) shows the cost estimation architecture for the restricted tuning session of this section. For every query there is a setup phase, where the single optimal plan is obtained through an optimizer call with a representative configuration. The representative configuration could contain any set of indexes satisfying the non-join or non-interesting order column restrictions (we could even use an empty configuration). The resulting internal subplan is saved in the *INUM Space*, which is the set of optimal plans maintained by the INUM.

Whenever we need to evaluate the query cost for some input configuration C, we use the *Index Access Cost Estimation* module to estimate the cost of accessing the indexes in C. The sum of the index access costs for C is added to that of the internal subplan to obtain the final query cost.

We assume that the *Index Access Cost Estimation* module is implemented by interfacing to the corresponding estimation modules of the optimizer. Computing only the individual index access

costs is much faster than a full-blown optimizer call, and does not affect reuse efficiency. Assuming additional optimizer interfaces does not limit INUM's flexibility. There are other ways to obtain index access costs, for instance through reverse-engineering the optimizer's analytical cost models. Our evaluation of the INUM with a commercial query optimizer uses pre-computed index access costs, which are obtained by invoking the optimizer with simplified, "sample" queries.

3.2.3 INUM Overview

Unlike the scenario of Section 3.2.1, in a real index selection session the "no join column" restriction is invalid, as we would typically consider indexes on join columns. The key difference with the previous section is that the assumptions supporting Theorem 3.2.1 are not valid and thus there might exist more than one optimal plan for a given query.

Caching Multiple Optimal Plans

Ignoring the applicability of multiple feasible plans (with different join orders and algorithms) as a function of multiple index configurations results in inaccurate cost estimates. To see why, consider the example query of Section 3.2.1 and assume a configuration C_1 with indexes on $T_1.ID$ and $T_2.ID$. The optimal plan for C_1 first joins T_1 and T_2 using a merge join and then joins the result with T_3 using a hash join. Now assume a configuration C_2, with indexes (other than $T_1.ID$ and $T_2.ID$) on T_2 and T_3. Now the merge join between T_1, T_2 is only feasible by inserting a sort operator on T_1. Existing approximation techniques [11, 12] will perform such insertions, ignoring the fact that indexes on T_2 and T_3 may favor an alternative plan that joins T_2 and T_3 with a merge join, without requiring additional sort operators.

To accommodate multiple optimal plans per query, we introduce the concept of the *INUM*

Space, a set that contains, for each query, a number of alternative execution plans. Each plan in the INUM Space is optimal for one or more possible input configurations. The INUM Space is essentially a "cache", containing plans that can be *correctly* reused to derive query costs. To guarantee correctness, we require the two properties defined below.

Definition 3.2.3 *The* INUM Space *for a query Q is a set of internal subplans such that:*

1. *Each subplan is derived from an optimal plan for some configuration.*

2. *The INUM Space contains* all *the subplans with the above property (i.e., derived from the optimal plans).*

According to Definition 3.2.3, the INUM Space will contain the optimal plan for *any* input configuration. Reusing that optimal plan results in accurate cost estimates *without* invoking the optimizer.

The key intuition of this chapter is that during the operation of an index design tool, the range of different plans that could be output by the optimizer will be much smaller than the number of configurations evaluated by the tool. In other words, we take advantage of the fact that an index design tool might consider thousands of alternative configurations for a query, but the number of different optimal plans for that query is much lower. For example, plans that construct huge intermediate results will never be optimal, and thus are not included in the INUM Space.

In addition, the optimality of a plan does not change very easily by changing index configurations, because it is determined by additional parameters such as intermediate result sizes. This chapter shows that the degree of plan reuse is high enough to justify the initial effort in obtaining the set of optimal plans by the huge number of optimizer calls that can be performed almost instantly afterward.

INUM formalizes the intuitive idea that if a plan is optimal for some configuration C_1, it might in fact remain optimal for a set of configurations that are "similar" to C_1. Notice that, contrary to previous techniques [11, 12], we use strict rules to determine the optimality of the reused plans, in the form of a *matching logic* that efficiently assigns, for each configuration input to INUM, the corresponding *optimal* plan.

System Architecture

Figure 3.3 (b) extends Figure 3.3 (a) with the modules required to implement the full INUM functionality. INUM takes as input requests from an index selection tool consisting of a query and a configuration for evaluation. The output is the optimal plan and cost for the query.

The *INUM Space* contains, for each query, the set of plans specified by Definition 3.2.3. The *Precomputation* module populates the INUM Space at initialization time, by invoking the optimizer in order to reveal the set of optimal plans that need to be cached per query. When invoking the INUM, the *Matching* module first maps the input configuration to its corresponding optimal plan and derives the query cost *without* going to the optimizer, simply by adding the cached cost to the index access costs computed on-the-fly.

In the remainder of the chapter we develop the INUM in two steps. In the first step we exclude from consideration query plans with nested-loop join operators, while allowing every other operator (including sort-merge and hash joins). We call such allowable plans MHJ plans. Section 3.2.6 extends our approach to include all join operators. Our two-step approach is necessary because nested-loop join operators require special treatment.

3.2.4 Using Cached MHJ Plans

We derive a formula for the query cost given an index configuration and use it to match an input configuration to its corresponding optimal MHJ plan.

A Formula for Query Cost

Consider a query Q, an input configuration C containing indexes $I_{T_1}..I_{T_n}$ for tables $T_1..T_n$ and an MHJ plan p in the INUM Space, not necessarily optimal for C. The reuse procedure of Section 3.2.2 describes how p is used with the indexes in C. Let c_{ip} be the sum of the costs of the operators in the subplan ip and s_{T_i} be the index access cost for index I_{T_i}. The cost of a query Q when using plan p is given by the following equation:

$$c_p = c_{ip} + (s_{T_1} + s_{T_2} + ... + s_{T_n}) \tag{3.1}$$

Equation 3.1 expresses the cost of any plan p as a ''function" of the input configuration C. It essentially distinguishes between the cost of the internal operators of a plan p (the internal-subplan, not necessarily optimal) and the costs of accessing the indexes in C. This distinction is key: An optimizer call spends most of its time computing the c_{ip} value. INUM essentially works by *correctly* reusing cached c_{ip} values, which are then combined to s_{T_i} values computed on-the-fly.

The following conditions are necessary for the validity of Equation 3.1.

1. c_{ip} is independent of the s_{T_i}'s. If c_{ip} depends on some s_{T_i}, then Equation 3.1 is not linear.

2. The s_{T_i}'s must be independent of p. Otherwise, although the addition is still valid, Equation 3.1 is not a function of the s_{T_i} variables.

3. C must provide the orderings assumed by Plan p. If a plan expects a specific ordering (for instance, to use with a merge join) but C does not contain an index to cover this ordering, then it is incorrect to combine p with C.

We can show that conditions (1) and (2) hold for MHJ plans through the same argument used in the proof of Theorem 3.2.1. The indexes in C are always accessed in the same way regardless of the plan's internal structure[2]. Conversely, a plan's internal operators will have the same costs regardless of the access methods used (as long as condition (3) holds). Note that the last argument does not mean that the selection of the optimal plan is independent of the access methods used.

Condition (3) is a constraint imposed for correctness. Equation 3.1 is invalid if plan p can not use the indexes in C. We define the notion of *compatibility* as follows:

Definition 3.2.4 *A plan is* compatible *with a configuration and vice-versa if plan p can use the indexes in the configuration without requiring additional operators.*

Assuming conditions (1)-(3) hold, Equation 3.1 computes query costs given a plan p and a configuration C. Next, we use Equation 3.1 to efficiently identify the optimal plan for an input configuration C and to efficiently populate the INUM Space.

Mapping Configurations to Optimal Plans

We examine two ways to determine which plan, among those stored in the INUM Space, is optimal for a particular input configuration: An exhaustive algorithm and a technique based on identifying a "region of optimality" for each plan.

Exhaustive Search Consider first the brute-force approach of finding the optimal plan for query Q and configuration C. The exhaustive algorithm iterates over all the MHJ plans in the INUM Space for Q that are compatible with C and uses Equation 3.1 to compute their costs. The result of the exhaustive algorithm is the plan with the minimum cost.

[2]Note that, even if partitioning is required for the hash joins, the equation remains valid as the selectivity (therefore the size of the data) remains constant during the physical design process

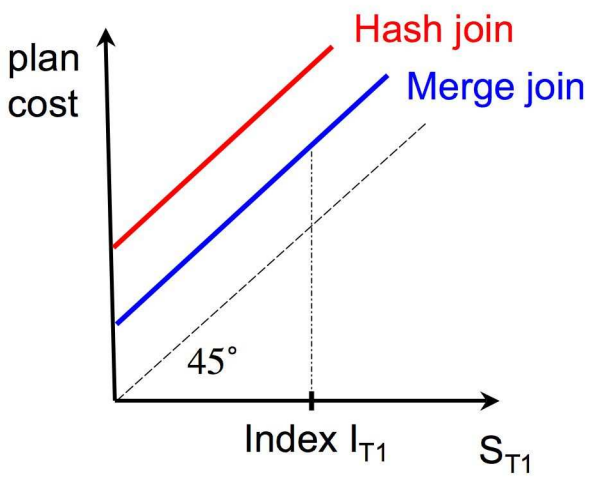


Figure 3.4: The cost functions for MHJ plans form parallel hyper-surfaces. The unit in the both the axes is the unit used by the optimizer to compare the costs of the plans. The slopes of the surfaces are determined by Eq. 3.1 to be 1.

The problem with the above procedure is that Equation 3.1 computes the total cost of a query plan *p if all the indexes in C are used*. If some indexes in C are too expensive to access (for example, non-clustered indexes with low selectivity), the optimal plan is likely to be one that does not use those expensive indexes. In other words, we also need to "simulate" the optimizer's decision to *ignore* an index. For this, the exhaustive algorithm needs to also search for the optimal plan for all the configurations $C' \subset C$ and return the one with the overall minimum cost. We call this iteration over C's subsets *atomic subset enumeration*.

If the INUM Space is constructed according to Definition 3.2.3, the exhaustive search with atomic subset enumeration is guaranteed to return correct results, but has the disadvantage of iterating over all the plans in the INUM Space and over all the subsets of C. In the next sections we show how to avoid the performance problems of the exhaustive search by exploiting the properties of Equation 3.1.

Regions of Optimality Consider a query Q accessing 2 tables, T_1 and T_2, with attributes $\{ID, a_1, b_1\}$ and $\{ID, a_2, b_2\}$. Q joins T_1 and T_2 on ID and projects attribute a_1 of the result.

Let C be a configuration with two indexes, I_{T_1} and I_{T_2}, on attributes $\{T_1.ID, T_1.a_1, T_1.b_1\}$ and $\{T_2.ID, T_2.a_2, T_2.b_2\}$ respectively. Let p_1 be a merge join plan that is the optimal MHJ plan for C. We ignore the subset enumeration problem for this example, assuming that we have no reason not to use I_{T_1} and I_{T_2}.

What happens if we change C to C_1, by replacing I_{T_1} with I'_{T_1}: $\{ID, a_1\}$? We can show that plan p_1 remains optimal and avoid a new optimizer call, using an argument similar to that of Section 3.2.2. Assume that the optimal plan for C_1 is p_2 that uses a hash join. Since the index access costs are the same for both plans, by Equation 3.1 the c_{ip} value for p_2 must be lower than that for p_1 and therefore p_1 cannot be optimal for C, which is a contradiction.

The intuition is that since both C and C_1 are capable of "supporting" exactly the same plans (both providing ordering on the ID columns), a plan p found to be optimal for C must be optimal for C_1 and any other configuration covering the same interesting orders. The set O of interesting orders that is covered by both C and C_1 is called the *region of optimality* for plan p. We formalize the above with the following theorem.

Theorem 3.2.2 *For every configuration C covering a given set of interesting orders O, there exists a* single *optimal MHJ plan p such that p accesses all the indexes in C.*[3]

Proof Let $C(O)$ be a set of configurations covering the given interesting order O. Also, consider the set P of all the MHJ plans that are compatible with the configurations in $C(O)$.

For every configuration C in $C(O)$ containing indexes on tables T_1,...,T_n we can compute the index access costs s_{T_1},...,s_{T_n} independently of a specific plan. Conceptually, we map C to an n-dimensional point $(s_{T_1}, s_{T_2}, ..., s_{T_n})$. The cost function c_p for a plan p in P is a linear function of

[3]Note that, if a plan does not use one of the indexes, then it is covered by a different set of interesting orders.

the s_{T_i} parameters and corresponds to a hypersurface in the (n+1)-dimensional space formed by the index access cost vector and c_p. To find the optimal plan for a configuration C, we need to find the plan hypersurface that gives us the lowest cost value.

By the structure of Equation 3.1, all hypersurfaces are parallel, thus for every configuration in $C(O)$ there exists a single optimal plan.

Figure 3.4 shows the cost hypersurfaces for a merge and a hash join plan, joining tables T_1 and T_2. To avoid 2-dimensional diagrams, assume we fix the index built on T_2 and only compute the plan cost for the indexes on T_1 that cover the same interesting order. The optimal plan for an index I_{T_1} corresponds to the hypersurface that first intersects the vertical line starting at the point I_{T_1}. Since the plan cost lines are parallel, the optimal plan is the same for all the indexes regardless of their s_{T_i} values.

The INUM Space exploits Theorem 3.2.2 by storing for each plan its region of optimality. INUM identifies the optimal plan for a configuration C by first computing the set of interesting orders O covered by C. O is then used to find the corresponding plan in the INUM Space. By Theorem 3.2.2 the retrieved plan will be the optimal plan that accesses all the indexes in C. As in the case of the exhaustive algorithm, to obtain the globally optimal plan, the above procedure must be repeated for every subset C' of C.

Atomic Subset Enumeration To find the query cost for an input configuration C we need to apply Theorem 3.2.2 for every subset of C and return the plan with the lowest cost. Enumerating C's subsets for n tables with an index for every table requires 2^n iterations. Since each "evaluation" corresponds to a fast lookup, the exponent does not hurt performance for reasonable n values. For $n = 5$, subset enumeration requires merely 32 lookups.

The overhead of subset enumeration might be undesirable for queries accessing 10 or 20 tables. For such cases we can avoid the enumeration by predicting when the optimizer will not use an index

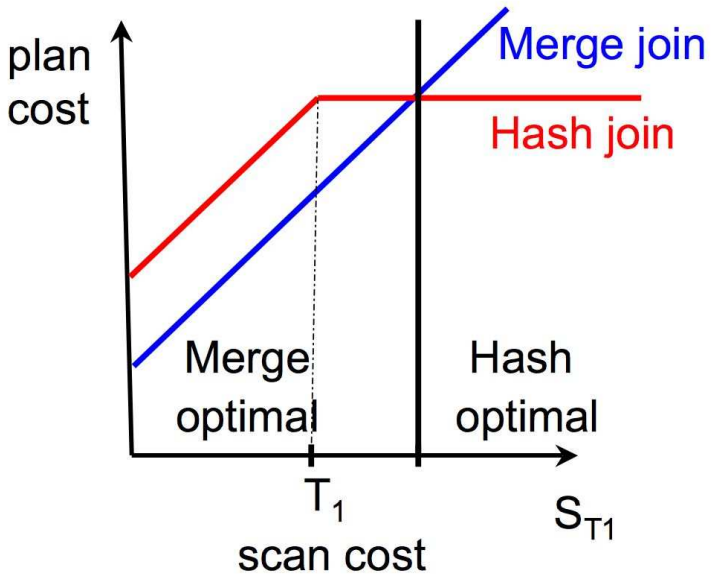


Figure 3.5: Modified plan comparison taking into account index I/O costs. The optimal plan for expensive indexes (to the right of the thick line) performs a sequential scan and uses a hash join.

of the input configuration C, or equivalently use a specific subset C'. Given a set of interesting orders, an index is not used only if it has an access cost that is too high. This can be proven by contradiction, as there is no reason why an index which provides the same interesting order, and has a higher access cost can be useful for a plan. If the index is not used for its interesting order, it is considered in the "empty" interesting order subplans. By storing with each plan the ranges of access costs for which it remains optimal, the INUM can immediately find the indexes that will actually be used.

Figure 3.5 shows an example of how the plan curves of Figure 3.4 change to incorporate index access costs. The hash join cost flattens after the index access cost exceeds the table scan cost (there is no need to access that index for a hash join plan). The hash join is optimal for indexes in the region to the right of the intersection point between the merge and hash join lines.

3.2.5 Computing the INUM Space

Theorem 3.2.2 in Section 3.2.4 suggests a straightforward way for computing the INUM Space. Let query Q reference tables $T_1, ..., T_n$ and let O_i be the set of interesting orders for table T_i. We also include the "don't care" interesting order in O_i, to account for the indexes on T_i that do not cover an interesting order.

The set $O = O_1 \times O_2 \times ... \times O_n$ contains all the possible combinations of interesting orders that a configuration can cover. By Theorem 3.2.2, for every member of O there exists a single optimal MHJ plan. Thus, to compute the INUM Space it is sufficient to invoke the optimizer *once* for each member o of O, using some representative configuration. The resulting internal subplan is sufficient, according to Theorem 3.2.2, for computing the query cost for any configuration that covers o. In order to obtain MHJ plans, the optimizer must be invoked with appropriate hints to prevent consideration of nested-loop join algorithms. Hints have the additional benefit of simplifying optimizer operation, because fewer plans need to be considered.

The precomputation phase requires fewer optimizer calls compared to optimizer-based tools, as the latter deal with different combinations of indexes, even if the combinations cover the same interesting orders. The number of MHJ plans in the INUM Space for a query accessing n tables is $|O_1| \times |O_2| \times ... \times |O_n|$. Consider a query joining n tables on the same *id* attribute. There are 2 possible interesting orders per table, the *id* order and the *empty* order that accounts for the rest of the indexes. In this case the size of the INUM Space is 2^n. For $n = 5$, 32 optimizer calls are sufficient for estimating the query cost for any configuration without further optimizer invocation.

For larger n, for instance for queries joining 10 or 20 tables, precomputation becomes expensive, as more than a thousand optimizer calls are required to fully compute the INUM Space. Large queries are a problem for optimizer-based tools as well, unless specific measures are taken to artificially restrict the number of atomic configurations examined [19]. Fortunately, there are ways to optimize the performance of INUM construction, so that it still outperforms optimizer-based

approaches. The main idea is to evaluate only a subset of O without sacrificing precision. We propose two ways to optimize precomputation, *lazy evaluation* and *cost-based evaluation*.

Lazy evaluation constructs the INUM Space incrementally, in sync with the index design tool. Since the popular greedy search approach selects one index at a time, there is no need to consider all the possible combinations of interesting orders for a query up-front. The only way that the full INUM Space is needed is for the tool to evaluate an atomic configuration containing n indexes covering various interesting orders. Existing tools avoid a large number of optimizer calls by not generating atomic configurations of size more than k, where k is some small number (according to [19] setting $k = 2$ is sufficient). With small-sized atomic configurations, the number of calls that INUM needs is a lot smaller.

Cost-based evaluation is based on the observation that not all tables have the same contribution to the query cost. In the common case, most of the cost is due to accessing and joining a few expensive tables. We apply this idea by "ignoring" interesting orders that are unlikely to significantly affect query cost. For a configuration covering an "ignored" order, the INUM will simply return a plan that will not take advantage of that order and thus have a slightly higher cost. Notice that only the c_{ip} parameter of Equation (3.1) is affected, and not the s_{T_i}'s. If an index on an "ignored" order has a significant I/O benefit (if for example, it is a covering index) the I/O improvement will still correctly be reflected in the cost value returned by the INUM. Cost-based evaluation is very effective in TPC-H style queries, where it is important to capture efficient plans for joining the fact table with one or two large dimension tables, while the joining of smaller tables is not as important.

3.2.6 Extending the INUM

In this section we consider plans containing at least one nested-loop join operator *in addition to merge or hash join operators*. We call such plans NLJ plans. We explain why plans containing nested-loop joins require additional modeling effort, and present ways to incorporate NLJ plans in

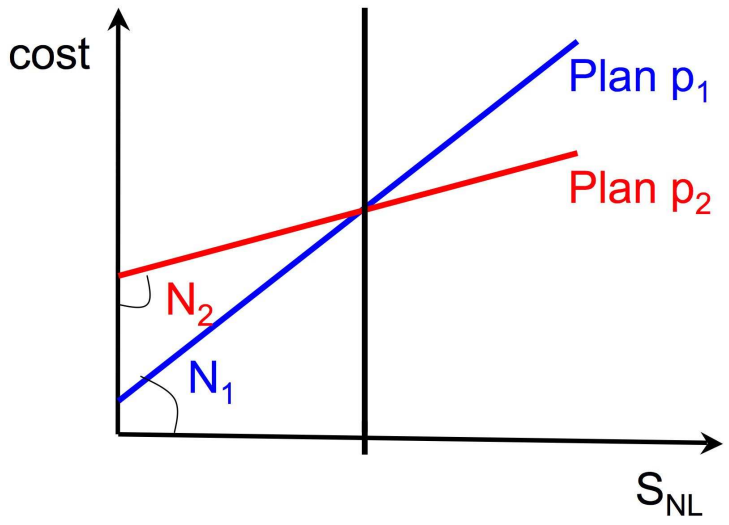


Figure 3.6: NLJ plan costs for a single table as a function of an index's s_{NL} parameter (System R optimizer).

the INUM.

Modeling NLJ Plans

The cost of an NLJ plan cannot be described by Equation (3.1) of Section 3.2.4. Therefore we can no longer take advantage of the linearity properties of Equation (3.1) for determining the plans that must be stored in the INUM Space and characterizing their regions of optimality.

We present an example based on System R's query optimizer [66] to illustrate the special properties of NLJ plans. Note that we do not rely or use System R's cost model in our system. The techniques in this section are not dependent on particular cost models, rather they capture the general behavior of NLJ plans. The System R example in this section is for illustration purposes only.

For System R the cost of a plan using index nested-loop join is expressed by $c_{out} + N \times c_{in}$, where c_{out} is the cost of the outer input, c_{in} is the cost of accessing the inner relation through index

I and N is the number of qualifying outer tuples.

c_{in} is given by $c_{in} = F \times (Pages(I) + Card(T)) + W \times RSI$, where F is the selectivity of the relevant index expressions, $Pages(I)$ is the index size and $Card(T)$ is the number of tuples in the table. W and RSI account for the CPU costs. It is easy to see that N and RSI are not independent of the plan, since both are determined by the number of qualifying outer tuples.

We define the *nested loop access cost* s_{NL} as $s_{NL} = F \times (Pages(I) + Card(T))$ and set $W = 0$ for simplicity. The nested-loop cost becomes: $c_p = c_{out} + N \times s_{NL}$.

Figure 3.6 shows the cost of different plans as a function of the nested-loop access cost for a single table. The difference with Figure 3.4 is that the hypersurfaces describing the plan costs are no longer parallel. Therefore for indexes covering the same set of interesting orders, there can be more than one optimal plan. In Figure 3.6, plan p_2 gets better than p_1 as the s_{NL} value increases, because it performs fewer index lookups. (lower N value and lower slope).

The System R optimizer example highlights two problems posed by the NLJ operator. First, it is more difficult to find the entire set of optimal plans, because a single optimizer call per interesting order combination is no longer sufficient. For the example of Figure 3.6, finding all the optimal plans requires at least two calls, using indexes with high and low s_{NL} values. A third call might also be necessary to ensure there is no other optimal plan for some index with an intermediate s_{NL} value. The second problem is that defining regions of optimality for each plan is not as straightforward. The optimality of an NLJ plan is now predicated on the s_{NL} values of the indexes, in addition to the interesting orders they cover.

In modern query optimizers, the cost of a nested-loop join operator is computed by more complicated cost models compared to System R. Such models might require more parameters for an index (as opposed to the s_{NL} values used for System R) and might have plan hypersurfaces with a non-linear shape. Determining the set of optimal plans and their regions of optimality requires exact knowledge of the cost models and potentially the use of non-linear parametric query opti-

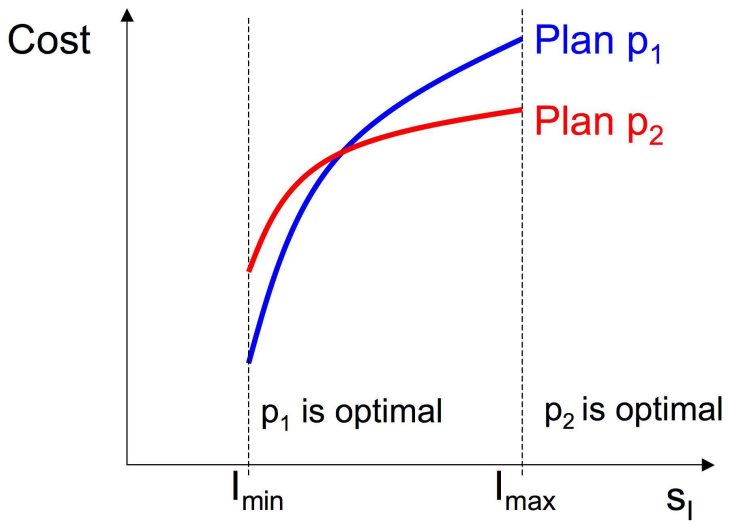


Figure 3.7: NLJ plan cost curves for a single table and an unknown cost function of a single index parameter s_I.

mization techniques [38]. In this chapter we are interested in developing a general solution that is as accurate as possible without making any assumptions about optimizer internals. The development of optimizer-specific models is an interesting area for future research.

Extending INUM with NLJ Plans

In this section we develop general methods for populating the INUM Space with NLJ plans in addition to MHJ plans, and for determining the overall optimal plan given an input configuration.

We begin with the problem of obtaining a set of optimal NLJ plans from the optimizer. We assume that each index is modeled by a single index parameter s_I (like the s_{NL} parameter in Section 3.2.6) that relates to the index's properties but we do not have access to the precise definition of s_I. The formula relating the s_I parameters to the plan costs is also unknown. Let I_{min} and I_{max} be two indexes having minimum and maximum s_I values respectively. We also assume that the plan's cost function is monotonically increasing, thus every plan has a minimum cost value for the most

"efficient" index I_{min} and maximum cost for I_{max}.

We present our approach using a simple example with a single table and a single interesting order. Figure 3.7 shows the plan costs for two different NLJ plans, as a function of a single index parameter s_I. Even without precise knowledge of the cost functions, we can retrieve at least two plans. Invoking the optimizer with I_{min} returns plan p_1, while I_{max} returns plan p_2. There is no way without additional information to identify intermediate plans, but p_1 and p_2 are a reasonable approximation.

Identifying the I_{min}, I_{max} indexes for a query is easy: I_{min} provides the lowest possible cost when accessed through a nested-loop join, thus we set it to be a covering index [4]. Using the same reasoning, we set I_{max} to be the index containing no attributes other than the join columns.

Performing 2 calls, one for I_{min} and for I_{max} will reveal at least one optimal NLJ plan. There are two possible outcomes.

1. At least one call returns an NLJ plan. There might be more plans for indexes in-between I_{max} and I_{min}. To reveal them we need more calls, with additional indexes. Finding those intermediate plans requires additional information on optimizer operation.

2. Both calls return an MHJ plan. If neither I_{min} nor I_{max} facilitates an NLJ plan, then no other index covering the same interesting order can facilitate an NLJ plan. In this case, the results of the previous sections on MHJ plans are directly applicable: By Theorem 3.2.2, the two calls will return the same MHJ plan.

For queries accessing more than one table, INUM first considers all interesting order subsets, just like the case with MHJ plans. For a given interesting order subset, there exists an I_{min} and an I_{max} index per interesting order. The INUM performs an optimizer call for every I_{min} and I_{max}

[4]There exist cases where I_{min} is a non-covering index, but in this case the difference in costs must be small. Generally, the covering index is a good approximation for I_{min}.

combination. This procedure results in more optimizer calls compared to the MHJ case, which required only a single call per interesting order combination. Multiple calls are necessary because every individual combination of I_{min} and I_{max} indexes could theoretically generate a different optimal plan.

We reduce the number of optimizer calls during NLJ plan enumeration by caching only a single NLJ plan and ignoring the rest. Instead of performing multiple calls for every I_{min}, I_{max} combination, INUM invokes the optimizer only once, using only the I_{min} indexes. If the call returns an NLJ plan, then it gets cached. If not, then INUM assumes that no other NLJ plans exist. The motivation for this heuristic is that a *single* NLJ plan with a lower cost than the corresponding MHJ plan is sufficient to prevent INUM from overestimating query costs. If such a lower NLJ plan exists, invoking the optimizer using the most efficient indexes (I_{min}) is very likely to reveal it.

Selecting the optimal plan for an input configuration when the INUM Space contains both MHJ and NLJ plans is simple. The optimal MHJ plan is computed as before (Section 3.2.4). If the INUM Space also contains an NLJ plan, the index access costs can be computed by the optimizer separately (just like for an MHJ plan) and added to the cached NLJ plan cost. INUM compares the NLJ and MHJ plans and returns the one with the lowest cost.

Modeling Update Statements

The INUM can readily be extended to estimate the cost of update statements (SQL *INSERT*, *UPDATE* and *DELETE*). An update can be modeled as two sub-statements: The "select" part is the query identifying the rows to be modified, and the "modify" part is the actual data update. The former is just another query and can be readily modeled by the INUM. The cost of the latter depends on factors such as the number of updated rows, the row size and the number of structures (indexes, materialized views) that must be maintained. Similarly to the individual *index access costs*, obtaining the cost of an update operation is simply a matter of interfacing to the relevant

optimizer cost modules and involves a simple, inexpensive computation.

Note that from a cost estimation perspective, handling updates is simple. The impact of updates on the *design algorithms themselves* and on their solution quality is an extremely interesting research topic, but beyond the scope of this chapter, that focuses only on cost estimation.

Handling Parameterized Queries

Often the query is provided to the optimizer with several values replaced by parameters. The optimizer optimizes the parameterized query by observing the distribution of the selectivity and finding a "robust" plan that behaves uniformly for all parameter values. Since the optimization process settles on a unique plan, INUM is able to use the plan as the basis to build the subplans. Therefore, presence of parameterized queries do not require special handling in INUM.

Extending to Partitions and Materialized Views

The above technique can be used to extend the algorithm for any design feature on which indexes can be built. The partitions and the materialized views are these type of features. INUM enables fast computation of the index reuse, but the plans corresponding to the other design features can be reused as well. We validate this claim by extending INUM in the presence of only partitions in chapter 5.

3.2.7 Experimental Setup

We implemented INUM using Java (JDK1.4.0) and interfaced our code to the optimizer of a commercial DBMS, which we will call *System1*. Our implementation demonstrates the feasibility of our approach in the context of a *real* commercial optimizer and workloads, and allows us to compare directly with existing index selection tools. To evaluate the benefits of INUM, we built on top

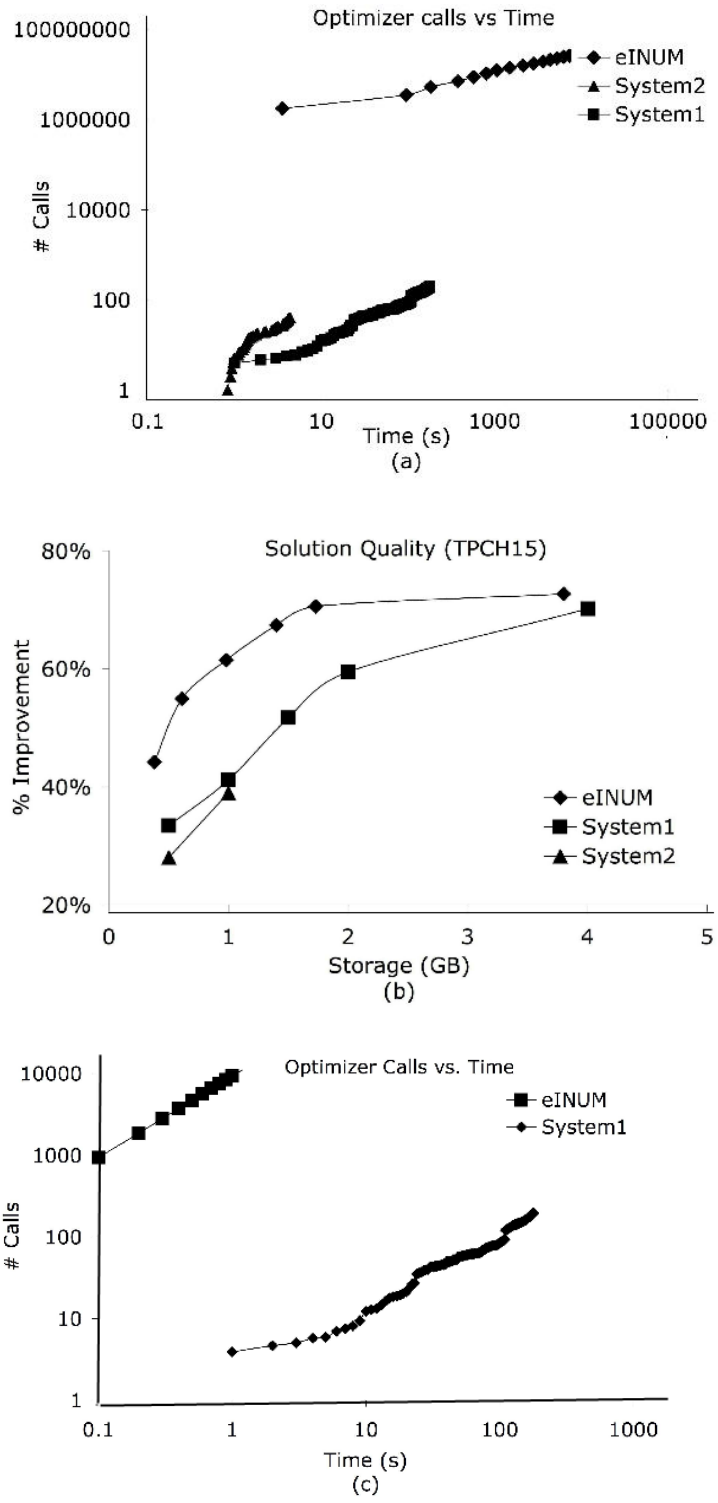


Figure 3.8: Experimental results for TPCH15. (a) Optimizer calls vs. time for an exhaustive candidate set (b) Recommendation quality (c) Optimizer calls vs. time for a heuristic candidate set

of it a very simple index selection tool, called *eINUM*. *eINUM* is essentially an enumerator, taking as input a set of candidate indexes and performing a simple greedy search, similar to the one used in [19].

We chose not to implement any candidate pruning heuristics, because one of our goals is to demonstrate that the high scalability offered by INUM can deal with large candidate sets that have not been pruned in any way. We "feed" *eINUM* with two different sets of candidate indexes. The *exhaustive* candidate set is generated by building an index on every possible subset of attributes referenced in the workload. From each subset, we generate multiple indexes, each having a different attribute as prefix. This algorithm generates a set of indexes on all possible attribute subsets, and with every possible attribute as key.

The second candidate set, the *heuristic*, emulates the behavior of existing index selection tools with separate candidate selection modules. We obtain *heuristic* candidates by running commercial tools and observing all the indexes they examine through tracing. The purpose of the heuristic candidate set is to approximate how INUM would perform if integrated with existing index selection algorithms.

Besides the automated physical design tool shipping with *System1*, we compare *eINUM* with the design tool of a second commercial DBMS, *System2*. We were unable to port *eINUM* to *System2* because it does not allow us to use index hints. Since we never actually ran *eINUM* with *System2*'s query optimizer, we cannot report on a direct comparison, but we include *System2* results for completeness. Integrating INUM with more commercial and open source database management systems is part of our ongoing work.

We experiment with two datasets. The 1GB version of the TPC-H benchmark [5] and the NREF

[5]We chose a relatively small version for TPC-H to speed-up administrative tasks such as building statistics and "real" indexes to validate our results. Dataset size affects the numerical values returned by the cost models but not the accuracy and speed of the INUM.

protein database described in [24]. The NREF database consists of 6 tables and consumes 1.5 GBs of disk space. For TPC-H, we used a workload consisting of 15 out of the 22 queries, which we call TPCH15. We were forced to omit certain queries due to limitations in our parser, but our sample preserves the complexity of the full workload. The NREF workload consists of 235 queries involving joins between 2 and 3 tables, nested queries, and aggregation.

We use a dual-Xeon 3.0GHz server with 4 gigabytes of RAM running Windows Server 2003 (64bit). We report both tuning running times and recommendation quality, which is computed using optimizer estimates. Improvements are computed by:

$$\%improvement = 1 - \frac{cost_{indexed}}{cost_{not\ \ indexed}}.$$

3.2.8 Experimental Results

In this section we demonstrate the superior performance and recommendation quality of *eINUM* compared to *System1* and *System2* for our TPCH15 and NREF workloads.

TPCH15 Results

Exhaustive Tuning Performance We provided *eINUM* with an exhaustive candidate set for TPCH15 consisting of 117000 indexes. For the exhaustive experiment, we ran all the tools without specifying a storage constraint. Figure 3.8 (a) shows the number of cost estimation calls performed by the 3 systems, and the time it took to complete them. The data for the two commercial systems come from traces of database activity. The horizontal axis corresponds to the time during which optimization is performed: for each point in the horizontal axis, the vertical axis of the graph shows the number of estimation calls up to that point in time. The graph focuses only on the tuning time spent during cost estimation, and not the overall execution time, which includes the algorithm itself, *virtual index construction*, and other overheads. Query cost estimation dominates

the execution time for all cases, so we discuss this first. We report on the additional overheads (including the time to construct the INUM model) later.

According to Figure 3.8 (a), *eINUM* performs the equivalent of 31 million optimizer (per query) invocations within 12065 seconds (about 3.5 hours), or equivalently, 0.3ms per call. Although such a high number of optimizer invocations might seem excessive for such a small workload, INUM's ability to support millions of evaluations within a few hours will be invaluable for larger problems.

Compare *eINUM*'s throughput with that of the state-of-the-art optimizer-based approaches (notice that the graph is in logarithmic scale). *System1* examines 188 candidates in total and performs 178 calls over 220 seconds, at an average of 1.2s per call. *System2* is even more conservative, examining 31 candidates and performing 91 calls over 7 seconds at 77ms per call. *System2* is faster because it does not use the optimizer during enumeration. However, as we see in the next paragraph, it provides lower quality recommendations. Another way to appreciate the results is the following: If we had interrupted *eINUM* after 220 seconds of optimization time (the total optimization time of *System1*, it would have already performed about 2 million evaluations!

The construction of the INUM took 1243s, or about 21 minutes, spent in performing 1358 "real" optimizer calls. The number of actual optimizer calls is very small compared to the millions of INUM cost evaluations performed during tuning. Also, note that this number corresponds to an experiment with a huge candidate set. As we show later, we can "compress" the time spent in INUM construction for smaller problems. *System1* required 246 seconds of total tuning time: For *System1*, optimization time accounted for 92% of the total tool running time. *System2* needed 3 seconds of additional computation time, for a total of 10 seconds. The optimization time was 70% of the total tuning time.

Exhaustive Tuning Quality Figure 3.8 (b) shows the recommendation quality for the three systems under varying storage constraints, where *eINUM* used the exhaustive candidate set. The

percentage improvements are computed over the unindexed database (with only clustered indexes on the primary keys). The last data point for each graph corresponds to a session with no storage constraint. *INUM*'s recommendations have 8%-34% lower cost compared to those of *System1*.

System2's unconstrained result was approximately 900MB, so we could not collect any data points beyond this limit. To obtain the quality results shown in Figure 3.8 (b), we implemented *System2* recommendations in *System1* and used *System1*'s optimizer to derive query costs. The results obtained by this method are only indicative, since *System2* is at a disadvantage: It never had the chance to look at cost estimates from *System1* during tuning. It performs slightly worse than *System1* (and is 37% worse than *eINUM* but the situation is reversed when we implement *System1*'s recommendation in *System2* (we omit those results). The only safe conclusion to draw from *System2* is that it fails to take advantage of additional index storage space.

We attribute the superior quality of *eINUM*'s recommendations to its larger candidate set. Despite the fact that *eINUM* is extremely simple algorithmically, it considers candidates that combine high improvements with low storage costs, because they are useful for multiple queries. Those indexes are missed by the commercial tools, due to their restricted candidate set.

Heuristic Enumeration In this section we demonstrate that using INUM in combination with existing index selection algorithms can result in huge savings in tuning time without losing quality. We use *eINUM* without a storage constraint, and we provide it with a candidate index set consisting of 188 candidate indexes considered by *System1*. *System1* was configured exactly the same way as in the previous session.

Figure 3.8 (c) shows the timing results for *eINUM* compared with *System1*, in a logarithmic plot. eINUM performs more query cost estimation calls (7440, compared to 178), yet cost estimation requires only 1.4 seconds, compared to the 220 seconds for *System1*. For a fair comparison, we must also take into account the time to compute the INUM Space. With *lazy precomputa-*

tion (Section 3.2.5), INUM construction took 180.6 seconds. Overall, *eINUM* took 182 seconds, compared to 246 seconds for *System1*. Note that *eINUM* does not implement any of the atomic configuration optimizations proposed in the literature for optimizer-based tools [19]. Incorporating additional optimizations would have reduced the precomputation overhead, since it would allow a further reduction in the number of optimizer calls.

The quality reached by the two algorithms was the same, which makes sense given that they consider exactly the same candidates.

INUM Accuracy INUM's estimates do not *exactly* match the query optimizer's output. Even the optimizer itself, due to various implementation details, such as variations in statistics, provides slightly different cost values if called for the same query and the same configurations. These slight differences exist between the plans saved by the INUM and the ones dynamically computed by the optimizer.

We measure the discrepancy E between the optimizer estimate for the entire workload cost c_{opt} and the INUM estimate c_{INUM} by $E = 1 - c_{INUM}/c_{opt}$. We compute E at the end of every pass performed by *eINUM* over the entire candidate set and we verify that the INUM cost estimate for the solution computed up to that point agrees with the "'real" optimizer estimate. We never found E to be higher than 10%, with an average value of 7%.

We argue that a 10% error in our estimate is negligible, compared to the scalability benefits offered by the INUM. Besides, existing optimizer-based tools that use *atomic configuration* optimizations [19] or the benefit assignment method for the knapsack formulation [33] already trade accuracy for efficiency.

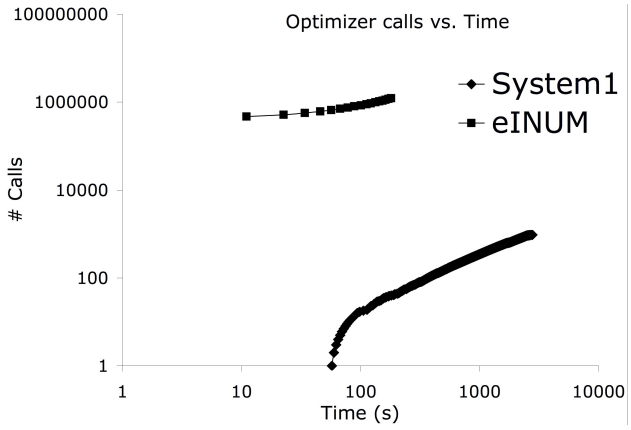


Figure 3.9: Optimizer calls vs. time for the NREF workload

NREF Results

In this section, we present our results from applying *eINUM* with an exhaustive candidate index set on the NREF workload. NREF is different from TPCH15 in that it contains more queries (235) that are simpler in terms of the number of attributes they access: Each query accesses 2 to 3 columns per table.

Figure 3.9 compares *eINUM* and *System1* in terms of the time spent in query cost estimation. *eINUM* performed 1.2M "calls", that took 180s (0.2ms per call). *System1* performed 952 optimizer calls that took 2700s (or 2.9s per call). INUM construction took 494s (without any performance optimizations whatsoever), while the total time for *System1* was 2800s. Interestingly, searching over the exhaustive candidate set with *eINUM* was about 6 times faster compared to *System1*, despite the latter's candidate pruning heuristics. We also compare the recommendation quality for various storage constraints, and find that *eINUM* and *System1* produce identical results. This happens because NREF is easier to index: Both tools converge to similar configurations (with single or two-column indexes) that are optimal for the majority of the queries.

3.3 Speeding up INUM

Although caching allows for three to four 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 offline designers, where the indexable structures such as materialized views and partitions are created dynamically, since a new cache must be built for every query using those structures.

We now 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. 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 used the dynamic programming-based optimization process of the PostgreSQL open-source DBMS query optimizer. Using INUM as the caching mechanism, we implemented PINUM, an index usage modeling technique for PostgreSQL. We obtained intermediate plan evaluations piggy-backed to the answer to each what-if question. We chose PostgreSQL because of its relatively mature query optimizer. We first implemented 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 found that the additional information reduces INUM's cache-building costs by a factor of 5 to 10. We then integrated the cache-based query cost estimation with a simple index selection tool to suggest indexes that speed up simple

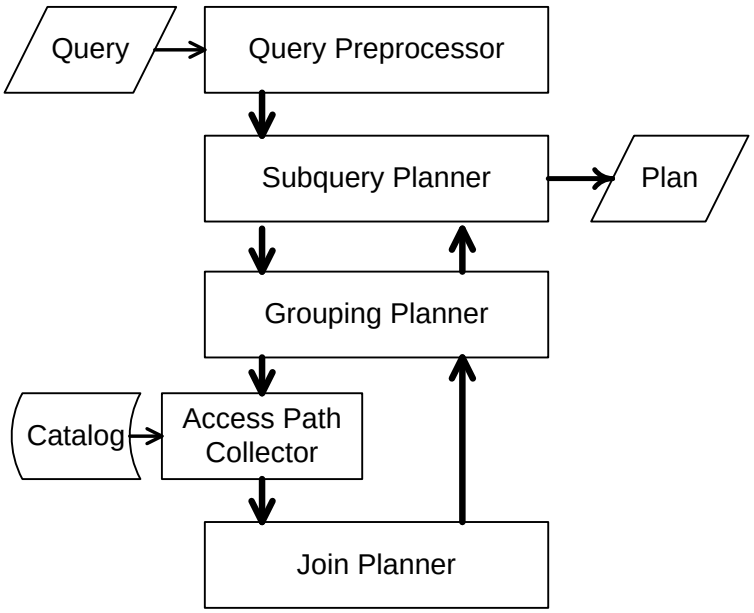


Figure 3.10: The PostgreSQL optimizer's architecture

analytical queries by factor of 10.

The rest of the section is organized as follows: We provide the necessary background PostgreSQL query optimizer in Section 3.3.1. Section 3.3.2 analyses the inefficiencies in the current approach, and Section 3.3.3 discusses PINUM's extensions and their implementation on the query optimizer. We discuss the experimental results using PINUM in Section 3.3.4.

3.3.1 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 [7].

Figure 3.10 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 opti-

mizes 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.

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.

3.3.2 Harnessing 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 5^{th} query in the TPC-H benchmark. The query joins 6 tables in the benchmark, and groups and orders the results. Since the join, group-by, and order-by clauses contribute to the interesting orders, and the interesting orders add up combinatorially, the query has 648 possible 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,?). 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 memory) overhead, as the optimizer builds 648 plans and transfers them to the client. Section 3.3.3 describes a pruning technique to reduce this overhead and output only the 64 useful plans.

3.3.3 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.

What-If Indexes

To determine the optimal plans in the 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.

Porting INUM to PostgreSQL

As Section 3.2.3 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.

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.

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 $S_A \subset S_B$ and $Cost(S_A) < Cost(S_B)$, then we remove Plan B.

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. This process is looks similar to the order reduction mechanism proposed by Simmen et al. [67]. The difference, however, is that we try to keep the maximum number of interesting orders instead of eliminating them to use the least number of indexes.

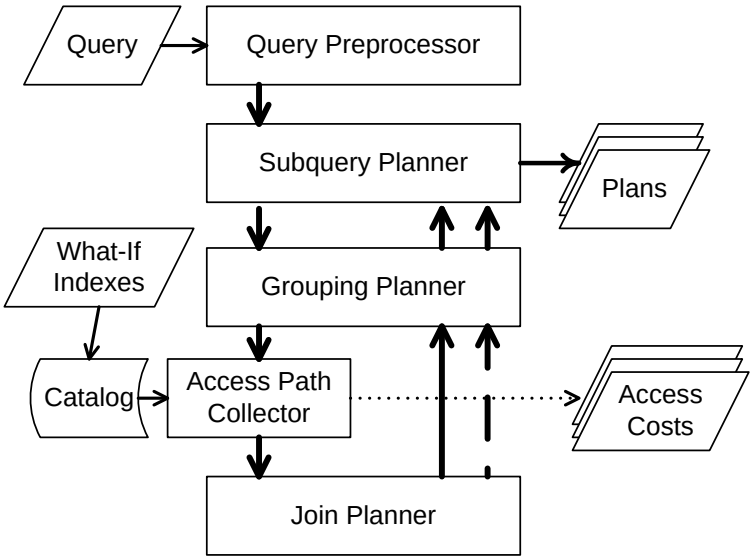


Figure 3.11: The modified query optimizer architecture. The dotted and dashed lines represent the new data flow induced by PINUM's access cost and cache construction optimizations.

Figure 3.11 shows the architecture of the query optimizer after the modifications for what-if indexes, fast access cost lookup and fast cache construction are added. As the figure shows, the changes to the optimizer components are minimal and require modifying only three files in the optimizer codebase.

3.3.4 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.

Experimental Setup

We implemented 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 behavior of the physical designer in the 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. We select a 10GB database for this experiment to build realistically sized dimension tables after the 1st level. 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 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 favors 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.

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. This 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.

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. Three queries had about a 4% error, and only one query had a 9% error in cost estimation. This demonstrates PINUM's cache-based cost model provides higher accuracy compared to INUM's cache-based cost model which has a 7% error on average. For the single poorly performing query, PINUM returns 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 the future, we intend to investigate the addition of these access paths to improve the model's accuracy.

Performance Results

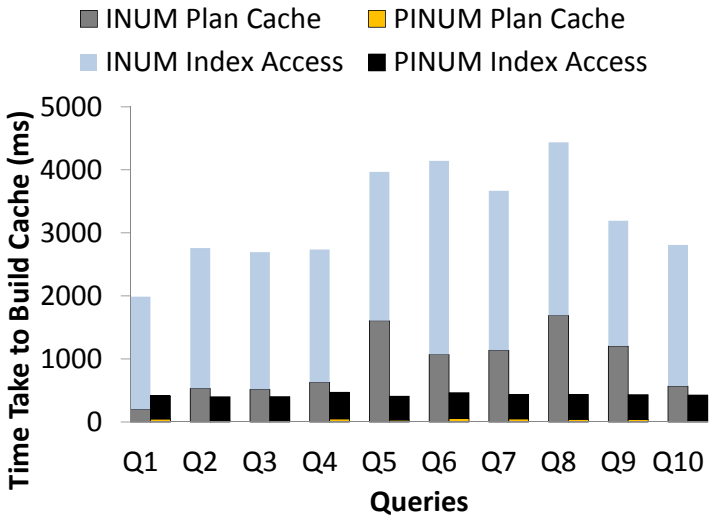


Figure 3.12: Comparison of cache construction times. Note that the time taken to build the PINUM plan cache is close to zero for each query.

Figure 3.12 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, and the y-axis shows the time taken to construct 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. We are currently implementing the aforementioned strategies.

Results for the Index Selection Tool

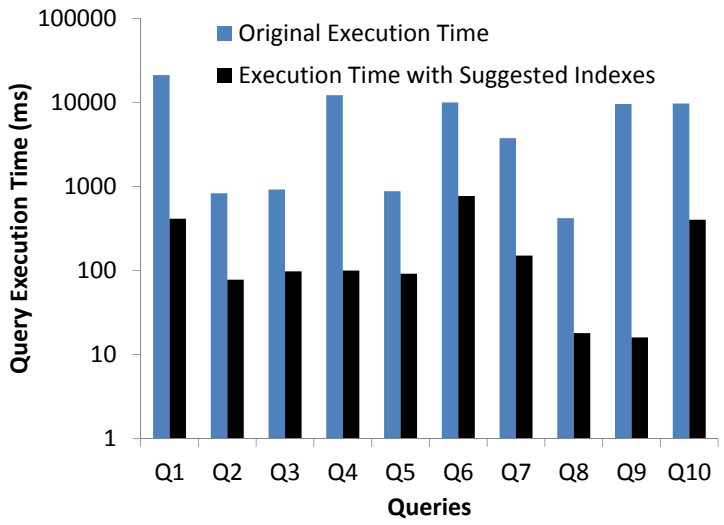


Figure 3.13: Workload performance improvement by using the index selection tool.

Figure 3.13 demonstrates the benefit of running the prototypical index selection tool (eINUM) discussed in Section 3.2.7. We ran the tool using the 10 queries in the workload, and restricted the tool to require 5GBs of space on disk for the suggested indexes. 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 non-covering 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 TPC-H.

3.4 Applications of INUM's Cost Model

INUM's cost model has the obvious application for index selection problem, as we discuss in this chapter and the next. Other than index selection, the model can be used to identify other interesting properties of the indexes, such as their correlation. Finding the correlations between indexes allows the DBA to determine how to materialize a set of indexes, so that they provide the maximum benefit during their materialization. In another instance, the correlation allows the DBA to remove indexes which negatively correlate with each other. In this section we describe how the correlation is computed and how to speed it up using INUM's cost model.

3.4.1 Computing Index Interactions

Definition 3.4.1 *For any pair of structures S_i and S_j we define the symmetric correlation coefficient $v_{ij} \equiv v_{ji}$ that represents the combined usage of S_i and S_j in executed query plans.*

Requirements from the Correlation Coefficient

Proposition 3.4.1 *The correlation coefficient v_{ij} should satisfy the following requirements:*

R_1 *v_{ij} is negative if S_i can replace S_j, positive if they collaborate, and approximately zero if they are used independent of each other in query plans.*

R_2 *v_{ij} can be normalized for any pair of S_i and S_j.*

R_3 *v_{ij} is easy to compute.*

Justification 1 *R_1: We justify by example.*

Example 3.4.1 *In a workload with only one query, Q =* `select` *A* `from` *T* `where` *B =* `'b'` `and` *C =* `'c'` *, the columns B and C should have positive correlation, while the indexes $I_{A-D} =$*

$T(A,B,C,D)$ and $I_{A-E} = T(A,B,C,D,E)$ should have negative correlation, and an irrelevant to the query index $T(E,F)$ should have zero correlation . It is straightforward that the pricing scheme requires these properties from the correlation coefficients V.

R_2: Without normalization, the correlations will be skewed towards the heavy queries, and the correlations on the light queries will be minimal.

R_3: It is necessary to compute all correlation coefficients V. Materialization and selection of the structures is performed for each query execution. Therefore, the correlation coefficients must be computed efficiently and scalably.

With respect to these requirements, we discuss a recently proposed correlation measure and its limitations. Then we propose a new measure that satisfies all the requirements.

Limitations of the Existing Approaches

Recently Schnaitter et al. proposed a technique that computes the correlation between indexes [64]. This section lists the limitations of this approach.

Given a set of indexes $I \subseteq S$ and two indexes from the set, $\{S_i, S_j\}$, their correlation coefficient v_{ij}^q given a query q, is:

$$v_{ij}^q = \max_{X \subseteq I, \{S_i,S_j\} \setminus X} \frac{co_q(X) - co_q(X_i) - co_q(X_j)}{co_q(X_{ij})} + 1 \tag{3.2}$$

Where, co_q is a function that gives the cost of q given a set of indexes. The set X is a subset of I that does not contain the two indexes S_i and S_j. Moreover, $X_i \equiv X \cup \{S_i\}$, $X_j \equiv X \cup \{S_j\}$, and $X_{ij} \equiv X \cup \{S_i, S_j\}$. The above measure finds the maximum benefit that an index gives compared to another index for a given query and any subset of the set, normalized by the total cost of the query

using both indexes. Since the query cost is monotonic, it is necessary that $co_q(X) > co_q(X_i) > co_q(X_{ij})$, $co_q(X) > co_q(X_j) > co_q(X_{ij})$.

Measure 3.2 does not satisfy the requirement R_1: for indexes that can replace each other the correlation is not negative. Since $co_q(X) > co_q(X_i) \approx co_q(X_j) \approx co_q(X_{ij})$, the measure is positive when the indexes are similar. It does not satisfy R_2 too: the produced values do not range in a bounded domain, therefore it is hard to perform normalization. Finally, it does not satisfy R_3: determining the coefficient requires exponentially large number of expensive optimizer calls even for a small I.

Structure Correlation Measure

We propose correlation measures that overcomes the limitations of the above technique. For indexes, we propose the measure:

$$v_{ij}^q = \frac{co_q(\{S_i\}) + co_q(\{S_j\}) - 2 \cdot co_q(\{S_i, S_j\})}{co_q(\{\}) - \min_{\{a,b\}} co_q(\{a,b\})} - 1 \tag{3.3}$$

Measure 3.3 identifies the individual benefits that the indexes S_i and S_j provide, and normalizes their sum w.r.t. the maximum benefit achievable by any pair of indexes $\{a,b\}$.

Proposition 3.4.2 *Measure 3.3 satisfies the requirements $R_1 - R_3$.*

Justification 2 *R_1: We show that R_1 is satisfied by proving its satisfaction for the extreme cases of structure collaboration and competition.* Case 1: *If S_i and S_j do not co-exist in query plans, then let us assume that S_i is very beneficial to a query q, hence $co_q(X_i) \to 0$ and S_j has no effect on it, hence $co_q(X_j) \to co_q(\{\})$. Since the cost function is monotonic [64], $co_q(X_{ij}) = co_q(X_i) = \min_{\{a,b\}} co_q(\{a,b\}) \to 0$. Hence, $v_{ij} \to 0$.* Case 2: *If S_i and S_j collaborate tightly in the extreme case, $co_q(X_i) = co_q(X_j) \to co_q(\{\})$, but $co_q(X_{ij}) \to 0$. Then, $v_{ij} \to 1$.* Case 3: *If the indexes are the same, then $co_q(X_j) = co_q(X_i) = co_q(X_{ij})$, implying that $v_{ij} = -1$.*

R_2: Since the cases discussed above are extreme, all structure correlation cases fall between them and, therefore their value is bounded by $[-1,1]$.

R_3: Using INUM we can ensure efficient computation of the correlation coefficients.

For columns, we propose the following measure:

$$v_{ij}^{q} = \begin{cases} 1 & \text{if } S_i \neq S_j \text{ and both used in q} \\ -1 & \text{if } S_i = S_j \text{ and used in q} \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

If two distinct columns appear in the same query, then they collaborate, otherwise they do not. Self-correlation for a column is set to -1, as a column can replace itself.

For a pair of index S_j and column S_i, we use the following measure:

$$v_{ij}^{q} = \begin{cases} 1 & \text{if } S_j \notin S_i \& \text{ both can be used in q} \\ -1 & \text{if } S_j \in S_i \& \text{ both can be used in q} \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

The index and the column correlate if the index does not contain the column, and both are useful to the query. If the index contains the column then the column is redundant in presence of the index, therefore, they compete. Finally, if the above conditions are not satisfied, then they do not collaborate, therefore the coefficient is 0.

So far, we discussed correlation of structures w.r.t. a specific query. We extend the correlation computation for a workload. If v_{ij}^{q} is the correlation of S_i and S_j for query q, then the coefficient for an entire workload is:

$$v_{ij} = \frac{\sum v_{ij}^{q} co_q(\{\})}{\sum co_q(\{\})} \tag{3.6}$$

Measure 3.6 normalizes the coefficients by using the maximum cost of the query. This allows the "heavy" queries to provide more weight to the correlation coefficient, when compared to the "lighter" queries.

As the section shows computing the query cost, given an index is the most expensive task for the index interaction computation. Using INUM reduces this overhead by three orders of magnitude.

3.5 Related Work

They key elements of modern automated database design tools are introduced in [1, 19, 33]. They strongly advocate the tight integration of database design algorithms and the query optimizer, in order to ensure that the recommended designs do in fact reflect the "true" (at least, as perceived by the system) query costs. The INUM, presented in this chapter, addresses the downside of relying on query optimization: its large computational overhead and the aggressive heuristics required to minimize it.

[20] shows that index selection is a computationally hard problem and thus efficient exact solutions do not exist. Design tools utilize some variation of a greedy search procedure to find a design that satisfies resource constraints and is locally-optimal. An example is the *greedy(m, k)* heuristic introduced of [1, 19]. Another approach uses a knapsack formulation [33] that greedily selects candidates based on their benefit to storage cost ratio. The knapsack-based approach is extended by [20], where a linear programming technique is used for accurate weight assignment. [54] applies similar greedy heuristics, along with a genetic programming algorithm, to the design problem of table partitioning in a parallel SQL database.

The *Index Merging* [18] work extends the basic design framework with more sophisticated techniques for performing candidate selection through merging candidate indexes. More recent work [11] suggest combining candidate selection with the actual search, so that the partial search results can be used to generate more useful candidates through various transformations. Both studies highlight the importance of effective candidate selection algorithms, that do not omit "potentially interesting" design candidates. The INUM improves any algorithm for candidate selection

(and the subsequent search) by allowing large candidate spaces to be constructed and traversed efficiently.

Modern database design tools support additional design features such as materialized views [1, 75], table partitions [3] and multidimensional clustering [75]. Having multiple object types increases the complexity of design algorithms, because combining design features generates very large *joint* search spaces. INUM can be extended to handle physical design features other than indexes and we expect its performance benefits to be even more pronounced when dealing with larger search spaces.

The work on Parametric Query Optimization (PQO) for linear, piece-wise linear and non-linear functions [37, 38] studies the change in the optimal plan for a query under changing *numerical* parameters, such as predicate selectivities. The INUM has the same goal, only that now the changing parameter is the underlying physical design, which cannot be captured solely by numerical values. The PQO framework, however, is invaluable in dealing with complex optimizer cost functions, especially the non-linear ones described in Section 3.2.6.

[55] presents an empirical study of the plan space generated by commercial query optimizers, again under varying selectivities. Their results suggest that while the space of optimal plans determined by the optimizer for a query might be very large, it can be substituted by another containing a smaller number of "core" plans without much loss in quality, an observation very similar to our own (See Section 3.2.3). [30] presents a technique to avoid unnecessary optimizer calls, by sharing the same plan for multiple *similar* queries. They define query similarity based on a number of query features (such as query structure and predicates).

3.6 Conclusion

Index selection algorithms are built around the query optimizer, however the query optimization complexity limits their scalability. We introduced INUM, a framework that solves the problem of expensive optimizer calls by caching and efficiently reusing a small number of key optimizer calls. INUM provides accurate cost estimates during the index selection process, without requiring further optimizer invocations. We evaluated INUM in the context of a real commercial query optimizer and showed that INUM improves enumeration performance by orders of magnitude. In addition, we demonstrated that being able to evaluate a larger number of candidate indexes through INUM improves recommendation quality. Although this chapter focused on a greedy algorithm to suggest the indexes, the simplicity of INUM's cost model allows it be used it more sophisticated index selection algorithms. In the next chapter, we discuss one such algorithm that takes full advantage of the INUM's cost model.

Chapter 4

Offline Physical Design

4.1 Introduction

Automated physical design is a major challenge in building self-tuning database management systems. The complexity of physical design emanates from the requirement of searching through a potentially huge space containing many design features, such as indexes, partitions, and materialized views. To make matters worse, each design feature interacts with other features to add a new dimension to the search space. Furthermore, in the real world, the search algorithm needs to satisfy several user-specified constraints, such as requiring the solutions to occupy less than a certain disk budget or requiring low maintenance cost or a limited number of features per table.

Existing commercial physical design tools provide suggestions in a reasonable time by heuristically pruning the solution space, and ignoring the interactions between the design features. This, however, prevents them from satisfying two important requirements: predictability and generality. Since the physical design problem is NP-Hard [20], finding the exact optimal solution may require exponential time for any reasonably sized program. Therefore, the selection tool must predict the quality of the solution, i.e., the distance of the proposed solution from the optimal, allowing the

DBA to accept the non-optimal solution in exchange for efficiency. By using simple regression, the DBA can even estimate the execution time required to arrive at a solution of certain quality. This ability to predict the quality of the solution also helps detect infeasible constraints set by the DBA. Therefore, predicting the quality of the solution is an essential requirement in the presence of a complex workload and constraints. Existing approaches lack this quality, since they prune the search space heuristically, thus limiting their knowledge of the solution quality with respect to the globally optimal solution.

Similarly, physical design tools need to handle a multitude of real-life constraints. The tool must be generic enough to handle new constraint types without a complete rewrite from the ground up. Since the pruning mechanism of greedy algorithms needs to be redesigned to address new constraint types, the proposed greedy algorithms are not general.

In the area of Operations Research, combinatorial optimization problem (COP) formulations and the algorithms for their solutions have been used for efficiently and scalably solving problems in which the underlying features interact with each other. They also provide feedback on the distance of the solution from the optimal, thereby allowing the user to terminate the optimization process upon reaching a certain quality guarantee. Unlike the greedy algorithms, COP solvers are generic–they solve a multitude of new constraint types without changing the code. The usefulness of this approach, however, depends on the "*convexity*" (Chapter 2) property of the COP formulation. If the COP formulation does not satisfy this crucial property, the solvers cannot use polynomial algorithms to solve the convex sub-problems and gradually arrive at the globally optimal solution.

Modeling the physical design problem as a COP is not straightforward because of the convexity requirement. The state-of-the-art formulations for database physical design achieve the convexity property by enumerating all possible combinations of the candidate indexes, making it impossible to scale without heavy pruning. Any pruning before the actual search process reduces the

predictability and efficiency, as the solver can predict and find solutions only within the pruned sub-space. The efficiency is also affected, since before pruning away a feature combination, one needs to evaluate its benefits.

Our Approach and Contributions: In this chapter, we discuss a COP-based tool called CoPhy(**C**ombinatorial **O**ptimization for **Phy**sical Design), which does not prune the search space heuristically, thus allowing it to provide quality guarantees. CoPhy uses a novel COP formulation to scale the problem size linearly with the candidate features. It then solves the problem efficiently using mature techniques such as Lagrangian relaxation. In this chapter, we focus on indexes as the design features, since they are the most commonly used by the DBAs and preserve the difficulty of the complete physical design problem.

The intellectual contribution of the chapter is: there exists a compact convex COP formulation for the physical design problem, thereby making the problem amenable to the sophisticated and mature combinatorial optimization solvers.

From the point of view of system design, the chapter contributes: An efficient and scalable solver for the COP, which combines the existing state-of-the-art techniques from the combinatorial optimization area to achieve its performance goals. It also allows DBAs to reduce the execution time at the expense of the quality of the expected solution. For example, by allowing a 1% difference from the optimal solution, it reduces the solver's execution time by an order of magnitude. Demonstrates the design of a *portable* physical designer. The system achieves its portability by using only very thin layer of interaction between the optimizer and the physical designer. Our experimental results demonstrate the system's superior performance on two different commercial DBMSs.

Organization: The rest of the chapter is organized as follows: We discuss the related work in Section 4.2. Section 4.3 builds the COP for plan selection on a workload, and Section 4.4 adds constraints to the COP. Section 4.8 details the architecture of CoPhy and the optimization methods.

We discuss our experimental results in Section 4.9, and finally conclude in Section 4.10.

4.2 Related Work

Existing commercial techniques use greedy pruning algorithms to suggest a physical design [2, 33], and use the optimizer directly, thereby reducing their efficiency and predictability (as the optimizer is a black-box). Caprara et al. were the first to propose a COP approach to the index-selection problem, by modeling it as an extension of the *facility-location problem* (FLP) [15], enabling it to exhaustively search the features, instead of greedily searching them. Their formulation, however, assumes that a query can use only a single index. Papadomanolakis et al. extended their formulation to account for queries using more than one index and also model index update costs [51]. Kormilitsin et al. propose lagrangian relaxation techniques to solve the FLP formulation [43]. Heeren et al. describe an approximation solution based on randomized rounding, assuming a single index per query[35]. Their solution has optimal performance but requires a bounded amount of additional storage. Zohreh et al. extend the FLP formulation to use materialized views, and then provide heuristics to find the optimal physical design in an OLAP setting [71]. Their algorithm is tuned towards materializing data-cube views and small number of indexes on them. Our approach scales to an index set two orders of magnitude larger than those reported in their work. Since these techniques use the FLP formulation, they use heuristic pruning to reduce the problem size to practical level, a limitation we avoid by proposing a new problem formulation.

4.3 Index Selection COP

Formulating the index selection problem to a COP allows us to use sophisticated combinatorial optimizers. We first describe the index selection process without any constraints, and then use it to

add complex constraints on it.

4.3.1 Index Selection for a Query

Plan selection and index selection are mutually complementary processes, as selecting the optimal plans involve selecting the optimal indexes and visa versa. To find the COP for plan and index selection, consider a query Q with O possible interesting order combinations. For each interesting order combination, INUM caches multiple plans. Let P_{op} be the p^{th} plan cached for the o^{th} interesting order. The plan can be used only if the atomic configuration covers the plan's interesting order combination. Let indexes $I_1, \cdots, I_t$ cover these interesting orders on tables $T_1, \cdots, T_t$. Using the first observation of INUM, the cost for P_{op} is:

$$Cost(P_{op}) = IC(P_{op}) + \sum_{i=1}^{t} AC(I_i) \tag{4.1}$$

The internal cost function $IC(P_{op})$ represents the cost of join and aggregation etc., $AC(I_i)$ is the access cost function that determines the cost of accessing an index I_i.

The plan cost is the minimum cost using indexes that cover the interesting orders on the table, hence:

$$Cost(P_{op}) = \min_{\alpha_i}(IC(P_{op}) + \sum_{i=1}^{t} \sum_{I_i \in CI(P_{op},T_i)} \alpha_i AC(I_i)) \tag{4.2}$$

$$\text{such that: } \sum_{I_i \in CI(P_{op},T_i)} \alpha_i = 1, \ \ \alpha_i \in \{0,1\} \forall i \tag{4.3}$$

The covering-index function $CI(P_{op}, T_i)$ finds the set of indexes that *cover* the *interesting order* required by P_{op} on table T_i. The variable α_i is a binary indicator associated with index I_i. If $\alpha_i = 1$, then the index I_i is used in the plan. Constraints in Eq. 4.3 ensure that only one index is used in the query plan for each table. CoPhy models the table scans as "empty" index scans, therefore, the equation does not consider table scans as special cases. This is a convex program, since the objective and the constraints are linear.

The cost of the query is the minimum cost of all the plans in cached for the query. Therefore, the cost of the query Q_q can be found as:

$$Cost(Q_q) = \min_{op} Cost(P_{op}) \tag{4.4}$$

Here each instance of the minimization problem $Cost(P_{op})$ has an independent set of constraints. Hence, to find the minimum cost, we iterate over all plans P_{op} for the query and minimize $Cost(O_{op})$.

4.3.2 Index Selection for a Workload

While index selection for a workload looks similar to index selection for a query. Using the same formulation can lead to errors in the presence of constraints. The reason is that for a single query the plans are independent of each other, i.e., there can be only one optimal plan selected. In a workload, however, each query has a plan, and the costs have to be minimized simultaneously, requiring a new formulation.

For a workload W, Eqs. 4.2 and 4.3 can be extended in a straightforward manner to select plans for each query in the workload.

$$Cost(W) = minimize \sum_{Q_q \in W} Cost(Q_q) \tag{4.5}$$

$$= \min_{\alpha_{iq}} \sum_{Q_q \in W} (\arg\min_{\alpha_{iq}} IC(P_{opq}) + \sum_{i=1}^{t} \sum_{I_i \in CI(P_{opq}, T_i)} \alpha_{iq} AC(I_i)) \tag{4.6}$$

$$\text{such that: } \sum_{I_i \in CI(P_{opq}, T_i)} \alpha_{iq} = 1 \;\; \alpha_{iq} \in \{0,1\} \forall i \tag{4.7}$$

The plan P_{opq} is the p^{th} plan cached for the o^{th} interesting order for query Q_q. The indicator variable α_i is changed to α_{iq}; otherwise selecting an index for use in one query forces it to be used

in all other queries. Since minimizing over another set of minimizations is not a convex function, this program is not a convex program. To convert it to a convex program, we introduce a new indicator variable p_{opq} which is set to 1 if the plan P_{opq} is selected for query Q_q. Since the second term in the query cost does not change in the equations hence forth, it is denoted by the term CL_{opq} for improved readability.

$$CL_{opq} = \sum_{i=1}^{t} \sum_{I_i \in CI(P_{op},T_i)} \alpha_{iq} AC(I_i) \tag{4.8}$$

$$Cost(W) = \min_{\alpha_{iq}, p_{opq}} \sum_{Q_q \in W} p_{opq}(IC(P_{opq}) + CL_{opq})) \tag{4.9}$$

$$\text{such that: } \sum_{p} p_{opq} = 1 \text{ and } p_{opq} \in \{0,1\} \forall o,p,q \tag{4.10}$$

The minimization, along with the constraints, convert the plan selection problem for a workload into a special form of convex program called a "quadratic program", because the objective function becomes a quadratic function involving variables p_{opq} and α_{iq}. Although quadratic programs can be solved efficiently in polynomial time, we improve efficiency by converting them to a linear program, thereby removing the quadratic term $p_{opq}\alpha_{iq}$. This is achieved by adding constraints as follows:

$$\arg\min_{\alpha_{iq}, p_{opq}} \sum_{Q_q \in W} \left(p_{opq}\, IC(P_{opq}) + CL_{opq} \right) \tag{4.11}$$

$$\text{such that: } \sum_{I_i \in CI(P_{op},T_i)} \alpha_{iq} = p_{opq} \;\; \forall T_i \tag{4.12}$$

Eq. 4.12 ensures that the plan P_{opq} is selected for query Q_q only if at least one index covering the interesting order requirement for each table has α_{iq} set to 1. Eqs. 4.11 and 4.12 form a linear program that selects plans for the entire workload and the indexes required for those plans.

Summarizing, Eq.4.11 defines the objective function for the index selection problem of a workload, and Eqs. 4.7, 4.10, and 4.12 define the constraints. Analyzing the size of this problem, observe that for a workload of $|Q|$ queries and $|P|$ cached plans, if there are $|I|$ indexes in the candidate

set, then the size of the generated program is $O((|Q| \times |I|) + |P|)$. For a typical workload $|P| << |I|$ (as discussed in last chapter), since the number of columns used in the workload is much larger than the interesting orders. Also, $|Q| << |I|$, since every query can contribute to large number of indexes. Therefore, the program size grows linearly with $|I|$. If $|P|$ is large for a workload, CoPhy uses approximation techniques [51] to reduce the cache size with small and bounded sacrifice in INUM's accuracy.

4.4 Adding Constraints

Without constraints, the plan selection and index selection problems are relatively straightforward. The problem, however, becomes much harder when the DBA requires the tool to satisfy some constraints upon the selected indexes. Traditionally, the index storage space has been the only constraint for the index selection tools. We discuss the index size constraints in this section, and provide a more complete discussion on adding other types of constraints to the COP formulation in Section 4.5.

The size constraint (or the index storage space constraint) ensures that the final size of the indexes does not exceed a threshold M, or similar conditions based on the index sizes. To translate these constraints, a condition on the indicator variables α_{iq} is not sufficient, since the same index can be used in multiple queries. Solely using these variables causes the problem formulation to double-count the index size.

Therefore, we derive a new variable α_i which indicates the presence of the index in any of the query plans. The α_i variable is the *logical-Or* of all the individual α_{iq} variables for all i. The logical-Or operation is translated by adding the following constraint to the generated program:

$$\alpha_{iq} \leq \alpha_i \;\; \forall i, q \;\; \alpha_i \in \{0,1\} \;\; \forall i \tag{4.13}$$

To prove that these constraints actually work, we show that if an index I_i is used in any query

Q_q, then the α_i variable is set to 1. Since $\alpha_{iq} = 1$, and $\alpha_i > \alpha_{iq}$, the constraint $\alpha_i \in \{0,1\}$ forces α_i to be 1.

The size constraint is represented by using the new variable α_i as:

$$\sum_i \alpha_i size(I_i) \leq M \tag{4.14}$$

Sometimes DBAs also specify constraints on the update cost of the indexes. We model this situation by adding the update cost to the objective of the optimization algorithm. Let μ_i be the update cost for the index I_i, then we add the following term to the objective in Eq.4.11.

$$\sum_i \mu_i \alpha_i \tag{4.15}$$

4.5 Adding More Constraints

Recently Bruno et al. propose a language to specify constraints [14]. In this section, we demonstrate the generality of formulating the physical design problem into COP by translating Bruno et al.'s sophisticated constraints into the COP formulation. All constraints introduced in this section are linear in nature, hence do not violate the convexity of the COP. We group the constraints into four categories: index constraints, configuration constraints, generators, and soft constraints, and describe their translations.

4.5.1 Index Constraints

The DBA may restrict the selection of the indexes by specifying conditions on the size of the index, the columns appearing in the index, or the number of columns appearing in indexes. Hence, we sub-group these constraints into three categories: column constraints, index width constraints, and index size constraints.

Index Count Constraint: The DBA may restrict the number of indexes that needs to be selected for a given table. Reducing the number of indexes in a table reduces their management overhead. Let T_i is the table on which we restrict the number of indexes to N, and the function $Indexes(T_i)$ represents the set of indexes on this table. Then the following constraint in our COP represents the constraint that the DBA wants to introduce:

$$\sum_{I_{\in} Indexes(T_i)} \alpha_i \leq N \tag{4.16}$$

Column Constraint: This group of constraints specifies the presence or absence of a column in the solution set. For example, one column constraint requires that at least one index built on a table T_t contains a column C_c. This condition can be translated to COP by adding the following constraint to the program for every such column:

$$\sum_{contains(C_c)} \alpha_{iq} \geq 1 \tag{4.17}$$

Reusing the notation from Eq. 4.6, we denote an index I_i being used in a query Q_q as the binary variable α_{iq}. The function $contains(C_c)$ finds the set of indexes which contain the column C_c. There can be many variations of this constraint, such as: only the first column of the indexes is C_c, or the index has a set of columns etc. All these variations are translated by changing the function $contains(C_c)$.

Index Width Constraint: This group of constraints limits the index search space by the number of the columns they contain. The translation of these constraints is similar to the previous one, as shown below:

$$\alpha_{iq} width(I_i) \geq N \tag{4.18}$$

Where, N is the limit on the number of columns for an index, and $width(I_i)$ determines the width of the index I_i.

4.5.2 Configuration Constraints

Although the DBA can specify any constraint on the configuration space, typically she constrains only the *final configuration* or the *result configuration* by limiting the costs of the queries using it. For example, the DBA may want to make sure that the final configuration speeds up all queries in the workload by at least 25% compared to the initial configuration. We translate such constraints into our COP by reusing the cost for the query in Eq. 4.2 and adding the following constraint to our COP:

$$Cost(Q_q) \leq 0.5\, InitialCost(Q_q) \tag{4.19}$$

The function $InitialCost(Q_q)$ represents the cost of the query before running the index selection tool.

4.5.3 Generators

Generators allow the DBA to specify constraints for each query, index, or table, without specifically mentioning them. It is equivalent to *for-loops* in regular programming languages. For example, if the DBA wants to restrict the final query cost, she specifies the following constraint:

`FOR` Q_q `IN W`

`ASSERT` $Cost(Q_q) \leq 0.5\, InitialCost(Q_q)$

The syntax of the constraint is self-explanatory. This constraint is translated by adding constraints shown in Eq. 4.19 for each query in the workload. The generator can also contain *Filters* to limit the scope of the constraints. For example, the following constraint limits the number of columns for indexes only when they contain the column C_3.

`For All` I_i `WHERE` $Contains(I_i, C_3)$

`ASSERT` $NumCols(I_i) \leq 4$

The "WHERE" condition filters the indexes using a boolean condition. $NumCols(I_i)$ determines the number of columns in I_i and $Contains(I_i, C_3)$ determines that the index I_i contains the column C_3. This filter is translated by generating the constraints in Eq. 4.17 and 4.18.

The constraint language also supports aggregates, such as SUM, COUNT etc. If the aggregation does not violate the convexity of the constraints, then they are added to the list of constraints generated by the program. Many of the common aggregations such as SUM, COUNT do not violate the convexity property and can be translated in this manner.

4.5.4 Soft Constraints

The constraints discussed so far are termed as *hard* constraints. The final solution proposed by the design tool has to satisfy all hard constraints. Sometimes the DBA may want to specify a *soft* constraint, which should be satisfied to the extent possible, if not completely. The description of the soft constraints is similar to that of an objective function, hence, we convert these conditions into objectives in our COP.

For example, the following constraint requires that the solver should try to achieve the lowest possible cost for the workload, but a solution is still valid even when the cost is not 0. The "SOFT" keyword indicates constraints which can be violated by the solver. `SOFT ASSERT` $Sum(Cost(Q_q)) = 0$ In COP, we add the following objective for the constraint:

$$\min \sum Cost(Q_q) \tag{4.20}$$

With multiple soft constraints, the optimization program becomes an instance of a *multi-objective* optimization [10] program. In multi-objective optimization, the solver does not search for one optimal point; rather, a set of points in the solution space which are *un-dominated*. A solution is called un-dominated, if no other point exists in the solution space for which every objective is smaller. The set of such points are called "pareto-optimal" or "sky-line" points.

When all objectives are convex, the pareto-optimal solution can be determined by using the "scalarization" method [10]: if the solver desires to optimize a vector $O_1, O_2, \cdots, O_n$ of objectives together, it creates a vector $\lambda_1, \lambda_2, ..., \lambda_n$ with $\lambda_i > 0$. Using these constants the solver optimizes the following problem:

$$\min \ \lambda_1 O_1 + \lambda_2 O_2 + ... + \lambda_n O_n$$

Since we assume each O_i to be convex, the scalarized-objective also remains convex. Hence, the complexity of solving the problem does not change, in presence of soft constraints. By varying the values of λ_i, the solver finds the pareto-optimal surface of the solutions and the DBA can decide on the optimal point on the surface.

4.5.5 Extending the Constraint Language

So far, we have discussed how to translate the state-of-the-art constraint language into solvable combinatorial optimization programs. It is possible to translate even more difficult constraints into COP, for example, the DBA can add a constraint to restrict the index selection so that the first columns in two indexes on a table are not the same. The existing language does not allow such constraints, and it is difficult to implement such constraints in a greedy constraint solver, since selection of one index depends on the presence or absence of another. This type of sophisticated constraints is translated and solved in CoPhy by adding the following constraint:

$$1 = \sum_{IsFirstCol(C_c)} \alpha_i \ \forall \, C_c \in T_t \tag{4.21}$$

The function $IsFirstCol(C_c)$ finds all indexes which have C_c as the first column.

4.6 Workload as a Sequence

So far the workload we have considered was a set of queries. Therefore the solution is also a set of indexes. All of the indexes should be created at the same time to achieve the performance suggested by the design advisor. There are many instances for which converting the workload to a sequence of queries can benefit the performance of the database.

For example, let us consider a scenario, in which long-running reporting queries run at midnight on a large warehouse. During the daytime, most of the queries are ad hoc, very selective queries. Let us also assume that the updates are random and occur throughout the day. Ideally, the long-running queries will need covering indexes to perform in the most efficient way, and the selective daytime queries can use single-column queries.

In a set-based workload, the automatic designer will probably select a relatively thin set of indexes that keep the maintenance cost low and only partially speed up the long-running queries at midnight. In a sequence workload, the designer can keep "thin" indexes, with only small number of attributes, during the day and build "fat" indexes at night before running the long-running queries. To avoid the maintenance cost of the fat indexes, one can disable/remove them during the daytime. Therefore, extending the workload model to include sequencing information is useful, compared to keeping them as sets.

In this section, we discuss how to extend the COP to include sequence information, and then solving it to compare the benefit we can get by using the COP.

4.6.1 Problem Definition

We formulate the sequence over time by breaking the workload W into a sequence of s discrete steps. Each step of the sequence contains a set of queries S_i. The queries from the set S_{i+1} always run after the queries in the set S_i. Inside the set S_i, there is no given order of running the queries.

The task of the advisor is to suggest indexes for each step of the sequence. The advisor needs to consider the cost of building the indexes. For simplicity, it is assumed that cost of dropping the index is zero. This assumption is violated for the clustered indexes, but modeling the cost of building a clustered index is beyond the scope of this thesis, therefore we focus only on the non-clustered indexes.

For simplicity of discussion, we assume that the only constraint for the designer is the index size constraint.

4.6.2 The COP for Sequences

We relax the sequence information by allowing multiple queries to be present at each step in the sequence. Let step i in the workload contain query set S_i. Reusing the notations from Section 4.3:

$$minimize \sum_{t=1}^{s} Cost(S_t) + \sum_{i=1}^{|I|} \alpha_{it} B_i(t) \tag{4.22}$$

$$\text{Subject To:} \tag{4.23}$$

$$\alpha_{it} - \alpha_{i(t-1)} = 1 \implies B_i(t) = BuildCost(I_i) \tag{4.24}$$

$$\alpha it \in \{0,1\} \forall i,t \quad B_i(t) \geq 0 \forall i,t \tag{4.25}$$

Here, $BuildCost(I_i)$ represents the cost of building the index I_i. $Cost(S_t)$ is the cost of the queries running at step t, and α_{it} represents the presence of the index I_i at the time step t.

The equations try to minimize the total cost of the workload which is the total query costs, plus the cost of building any indexes. Constraint 4.24 ensures that the cost to build the index is factored in if the index was not present in the earlier step. This is a logical constraint, instead of a linear one. To convert it into a linear constraint, we replace it with the following one:

$$Cost(I_{build}) - B_i(t) + \mathrm{M}(\alpha_{it} - \alpha_{i(t-1)} - 1) <= 0 \tag{4.26}$$

Here M is a really large value. If $\alpha_{it} - \alpha_{i(t-1)} - 1 < 0$, then the expression on the LHS is a large negative number, therefore $B_i(t)$ can be made 0 without violating the constraint. Since the objective is to minimize, the value will be set to zero always. If $\alpha_{it} - \alpha_{i(t-1)} - 1 = 0$, then $B_i(t)$ has to be at least $Cost(I_{build})$. Since α_{it} are binary variables, other possibilities do not exist.

To estimate $BuildCost(I_i)$, we approximate the cost of building the index to sorting the columns and storing them back on disk. Hence,

$$BuildCost(I_i) = SortCost(I_i) + StoreCost(I_i)$$

If an index I_i is defined as $T(A,B)$, then we estimate the cost of reading the table data and sorting by asking the optimizer cost of the following query:

QI_i `= select A,B from T order by A,B`

There is no generic way to find the cost of storing a set of pages from the optimizer. Storing the data back, however, is a simple operation and the cost should be directly proportional to the number of pages stored. Therefore, we run experiments on the database for different page sizes, and then regress the data to find the linear coefficients of that regression. Putting everything together, if a, and b are the coefficients of the storing cost equation:

$$BuildCost(I_i) = Cost(QI_i) + a \times PageCount(I_i) + b$$

Solving the program gives us various values of α_{it}s. If $\alpha_{it} - \alpha_{i(t-1)} = 0$, then we do not take any action. If $\alpha_{it} - \alpha i(t-1) = -1$, then we drop the index, else we build the index at that time step. Note, that we assume that the indexes will be built before the arrival of the queries in the

next step. We can generalize the COP formulation to add more constraints on indexes to be built between the steps.

This is a special case for the online-algorithms, and we show the implementation and detailed experimental results for this formulation in Chapter 5.

4.7 Using C-PQO as Cost Model

So far in this chapter, we use INUM as the cost model for the COP. While INUM is open and flexible, it is slower and has more error than the cost model provided by C-PQO. C-PQO has full access to the optimizer, and can provide the exact cost returned by the optimizer, instead of providing 1-2% error, as in the case of INUM.

In this section, we describe how to build the cost model using C-PQO as the cost model. We first give the overview of C-PQO, and then convert the C-PQO model into a INUM-like linear model. The model can then be used to build a COP, just like the one we have described.

From a very high level, C-PQO builds a "super-plan" for each query in the workload. The super-plan is the union of all possible optimal plans for the query. For example, consider a very simple query Q:`select sum(A) from T where B = 'value'`. Figure 4.1 shows the plan super-plan generated by C-PQO.

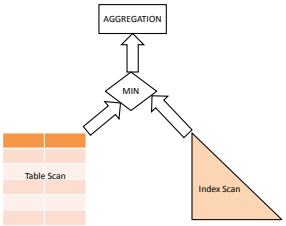


Figure 4.1: Example of the C-PQO plan for the query Q.

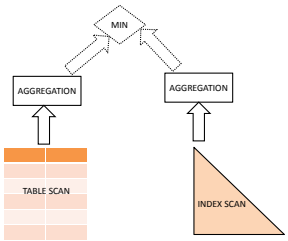


Figure 4.2: Example of the INUM plan for the query Q

Figure 4.2 shows the corresponding plans in INUM's cache. Note that the "Min" node on the top is an assumed minimum condition, since we do not put the condition explicitly in the cache. As the figure shows, the super-plan from C-PQO can be easily converted into the set of plans of INUM by traversing the C-PQO plan in post order and pushing the Min-nodes up to the root of the tree. At each point of pushing the Min-node, the converter needs to remember the imposed conditions on each side. For example, in Figure 4.1, the converter remembers that the RHS of the Min-node requires an index to be viable. These conditions then get translated to the function *CI* in Section 4.3's Equation 4.2. Thus, instead of returning just the covering indexes for the interesting orders, the function returns all the valid indexes for the plan in question.

Once the C-PQO plan is converted to the INUM's set of plans, the COP can be directly applied to it and solved efficiently as discussed before. Therefore the COP is generic enough to solve other fast cost models. In fact, as long as the underlying optimization form is convex, the COP can be used to solve such problems. Many classes of problems fall into this category, most importantly dynamic programming optimizers. Since, most of the modern query optimizers use the dynamic programming as the basis, the COP can be ported to them easily.

4.8 CoPhy System Design

In this section, we discuss the complete architecture for the index selection tool - CoPhy, including the process of selecting candidate indexes and techniques to solve the COP developed in the earlier sections.

Figure 4.3 shows the most important modules of CoPhy. All modules except the *Solver* are used for building the COP. These modules decide the α_{iq} and the p_{opq} variables and compute the *IC* and the *AC* functions. Once the COP is constructed, the *Solver* is used to determine the solution.

The *Candidate Selection* module determines the candidate indexes used for the α_{iq} variables,

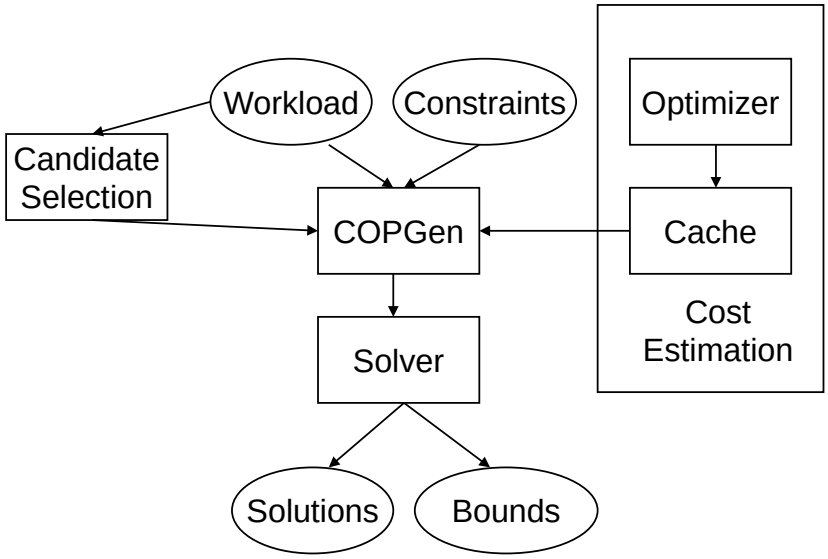


Figure 4.3: Architecture of CoPhy

using the workload's structural properties. The *COPGen* module takes these candidates and the constraints as input, and generates the COP. *Cost Estimation* determines the cost model for the queries, and our tool uses INUM as the cost model. This model caches the plans, the internal cost functions (*IC*s), and the access cost functions (*AC*s). The complete optimization problem is consequently input to the *Solver*. The overall performance of the tool depends on the number of α_{iq} and p_{opq} variables, which control the problem size and, consequently, the solver's execution time.

We use a very simple *Candidate Selection* module, which generates indexes for every subset of the columns referenced in a query, eventually producing a large set of candidate indexes. From this large candidate set, the module prunes out indexes that never help in reducing query costs. It identifies these indexes by finding the minimum access costs for all tables in the query. If the minimum cost of the query using a given index along with the best possible indexes on all other tables, is more than the cost of that query without that index, then that index will never be used by the optimizer. Generally, the module produces in the order of thousands of candidate indexes. In comparison, the commercial physical design tools consider only up to hundreds of candidate

indexes. As we demonstrate in Chapter 3, using larger set of candidate indexes provides better quality solutions.

Now we focus on the *Solver*, which is the critical part of CoPhy, as it provides the desired efficiency and scalability.

4.8.1 Solving the COP

The *Solver* module of Figure 4.3 takes as input the optimization formulation corresponding to the index selection problem, and computes the near-optimal solution. While optimization problems with integer variables are NP-hard in the worst case, mature solving techniques in practice efficiently optimize very large problem instances. This subsection discusses the details of the algorithms that enable a fast solution for such large problems. First, a fast greedy algorithm is proposed, then the complete solution is described.

Greedy Algorithm

We extend the eINUM iterative greedy algorithm proposed in [53] as a baseline solver for the COP. Each iteration of the greedy algorithm determines the index which decreases the workload cost the most and adds it to the solution set. The algorithm terminates when there are no more indexes to add or the storage constraint is violated. The eINUM algorithm however is impractical as a physical designer, since it takes about 3.5 hours to suggest indexes. We therefore extend the algorithm by using the structure of the INUM cache to speed it up by a factor of 300! The details of the eINUM algorithm and the optimizations are discussed in the next section.

Lagrangian Relaxation Algorithm

Generally the combinatorial optimization programs are solved using the *branch-and-bound* method. In this method, the solver starts with an initial upper bound on the program (In our case the solution of the greedy algorithm). For variable x in the problem, it creates two branches, one with $x = 0$ and the other $x = 1$. Before exploring a branch, the solver tests to see if the lower-bound of the branch is higher than current upper bound. If the lower bound is higher, then that branch is safely pruned and the search continues in other branches. If it has pruned out all the branches or all variables are integral variables, it backtracks to search the branches in the parent nodes.

Commercial COP solvers such as CPLEX use a linear programming (LP) relaxation to find the lower and upper bound of the branch-bound process. In an LP relaxation, the binary constraints are ignored and the problem is solved to find the lower bound of the objective. While LP relaxation works well with many problem formulations, including FLP [51], in our formulation, the linear relaxation of the α_{iq} variables allows them to be set to very small values. This effectively estimates the lowest possible cost for the workload without any constraint. For tight constraints, this lower-bound is far off the desired optimum. Therefore, CPLEX, which uses only LP relaxation, runs for hours before converging to the optimal solution for our COP. In this section, we improve the bound computation by using Lagrangian relaxation (LR) to compute the upper bound and a randomized-rounding method on the LP relaxation to compute the lower bound.

To understand the LR method to compute the upper or lower bounds, consider the example problem shown below

$$\textit{minimize } f(x)$$

$$\text{Such that: } g_1(x) \leq A \;\; g_2(x) \leq B$$

This problem has $f(x)$ as the objective function and includes many constraints. The Lagrangian relaxation identifies the "tough" constraint among the list of constraints and adds that to the objec-

tive function with a multiplier. Intuitively, it punishes the solver by a factor θ, if it does not satisfy the constraint. For example, if the first constraint is the toughest constraint in the problem, then the relaxed problem becomes

$$\textit{minimize } f(x) + \theta(g_1(x) - A)$$

$$\text{Such that: } g_2(x) \leq B \;\; \theta > 0$$

This modified problem may not satisfy the constraint $g_1(x) \leq A$, since it has been moved to the objective function. But from the solution found using this problem, we cascade the constraints, similarly to the technique in [43], to find the upper bound on the original problem. By finding an appropriate value of θ, the upper bound is made tighter and the solver converges to the solution faster.

There is no systematic method to find the tough constraints for a specific problem, since it depends on the problem structure. For COP, the constraint which combines the α_{iq} variables into α_i variable (constraint in Eq. 4.13) is the toughest constraint. Removing that constraint splits the problem into two components, each of which can be solved efficiently. The constraint, however, makes even sophisticated solvers take considerable time to solve the problem. Hence, we use that constraint as the "tough" constraint for the LR technique and then operate in the branch-bound method to solve the problem.

To find the lower bound on a node, CoPhy considers the solutions to the LP relaxed problem. If $0 < \alpha_{iq} = r < 1$, and for a random value $0 < s < 1$, we set $\alpha_{iq} = 1$ if $s > r$. This randomized rounding creates a solution which uses more indexes than allowed by the constraints, but has been shown to provide a tight lower bound on the cost function [35].

Since at each node CoPhy knows the upper and the lower bound on the objective value, it can gain speed by trading off the accuracy of the final solution by looking at the difference between the upper and lower bounds. This allows the DBA to terminate the optimization problem when

the difference goes below a given percentage and accept the results from that search step as the solution.

Furthermore, this allows the DBA to predict when a certain quality guarantee will be achieved by using simple regression. Thereby, she can estimate the quality of the solution after an estimated period of time, and intelligently predict when to terminate the optimization session. Since the COP is based on INUM's cost model, errors in INUM's estimation can limit the solver's knowledge of the exact optimal value. In our experiments, INUM has about a 7% error in cost estimation, which we argue to be reasonable for the scalability it provides. Moreover, it is an orthogonal problem to improve INUM's accuracy using more cached plans.

4.8.2 Greedy Algorithm

In this section, we discuss a fast greedy algorithm to find a candidate final configuration. The greedy algorithm is fast because it always considers only one index at a time, i.e., does not consider the index interaction effects. It still provides satisfying results, however, because it considers a large number of candidates.

Alg. 1 shows the algorithm, which consists of two loops, each selecting a candidate solution set. Line 4 finds the index I_m such that, if added to the set of selected indexes, provides the maximum reduction to the cost of the workload. The function $Cost(W,S)$ determines the cost of the workload with the index set S. The loop terminates when no new index satisfied the $sizeConstr$ constraint. Similarly, in the second loop, Line 11 finds the index with maximum improvement for the objective function per unit storage cost. The candidate solution sets for the indexes are called S_1 and S_2 respectively. Since S_1 does not consider the storage overhead of the indexes, it generally selects larger indexes. Selecting large indexes reduces the cardinality of the set S_1 as the storage limit will be hit sooner. The S_2 set on the other hand gives more weight to the index size and holds more indexes of smaller size each. The algorithm then returns the candidate set with the greater

Data: I: The candidate set of indexes

O: The objective function

$sizeConstr$: Index size constraint

Result: S: the solution index set

1 initialization:;

2 $S_1 = S_2 = \phi$;

3 **while** *true* **do**

4 $\quad I_m = \arg\max_{I_i \in I \setminus S_1} (Cost(W, S_1) - Cost(W, S_1 \cup I_i))$;

5 $\quad$ **if** $sizeConstr(S_1 \cup I_m)$ **then**

6 $\quad\quad S_1 = S_1 \cup I_m$;

7 $\quad\quad I = I - I_m$;

8 $\quad$ **else**

9 $\quad\quad$ break;

10 $\quad$ **end**

11 **end**

12 Repeat 3-11 with $I_{ms} = \arg\max_{I_i \in I \setminus S_1} \frac{Cost(W,S_2) - Cost(W,S_2 \cup I_i)}{Size(I_i)}$;

13 $return\ S = Max\_Benefit(S_1, S_2)$;

Algorithm 1: Greedy algorithm for sub-modular index-selection problem

improvement.

This algorithm directly selects the α_i variables in Eq. 4.13. By setting the dependent variables, such as α_{iq}, CoPhy determines the p_{opq} variables that are used in the objective function in Eq. 4.11. Therefore, this algorithm not only solves the index selection problem, but also determines the cost of the final objective by cascading the constraints.

Optimizing the Greedy Algorithm: In Alg. 1, finding I_m and I_{ms} dominates the execution time of the greedy algorithm. A naive implementation, which iterates over all possible configura-

tions to determine the benefit of the index table takes several hours for a 15 query workload [53].

We speed up the performance of this algorithm by building an in-memory lookup structure. The structure consists of two large hash tables. The first hash table, named *CostMap*, contains a mapping of the form $(P_{opq}, T_t) \rightarrow MinCost$. Reusing the notations in Section 4.3, P_{opq} represents the p^{th} plan of the o^{th} interesting-order combination for query Q_q, and T_t is a table used in the query. The value $MinCost$ is the current minimum cost for accessing data for the table T_t in the plan P_{opq}. The hash table is initialized with $MinCost = \infty$ for each entry. This hash table allows the greedy algorithm to easily find the current best solution for each cached plan. The index I_i is of interest only if it lowers the current minimum cost of any of the plans.

The second hash table, named *IntMap*, contains the mapping of the form $C_c \rightarrow Q_q \rightarrow List < Plans >$, where C_c is a column in the table, $List < Plans >$ is the list of plans which benefit by using column C_c as an interesting order, and Q_q is used to group all such interesting orders in query Q_q. For each new index I_i, if it covers the interesting order C_c, the greedy algorithm directly looks up the queries and plans benefiting from that interesting order.

Since the greedy algorithm needs to find the queries which benefit from an index, and the amount of benefit the index provides, using these two structures speeds up both the bottleneck functions by a factor of 300 (as shown in Section 4.9).

4.9 Experimental Results

This section discusses CoPhy's results on a TPC-H-like workload and on two mainstream commercial DBMS. We first discuss the experimental setup and then study the workload in detail. We start with the index storage constraints, and then discuss the effect of the quality guarantee on the execution time.

4.9.1 Experimental Setup

We implement our tool using Java (JDK1.6.0) and interface our code to the optimizer of a popular commercial DBMS, which we call *System1*.

We experiment with a subset of the TPC-H benchmark containing 15 queries (our experimental parser, for the time being, does not support the following queries: 7, 8, 11, 15, 20, 21, and 22). Since experiments with larger databases show trends similar to those in a 1GB database, we use 1GB database on all the workloads for ease and speed of result verifications. For these 15 queries, INUM caches 1305 plans and the candidate selection module generates 2325 indexes. These cached plans enable INUM to predict the plans with 7% cost estimation error. We also show the behavior of the system on other real-world and synthetic benchmarks in Appendix 4.9.6.

We use a dual-Xeon 3.0GHz based server with 4 gigabytes of RAM running Windows Server 2003 (64bit), but for our experimental purposes we limit the memory allocated to the database to only 1GB. We report both selection tool execution time and quality of the recommended solutions computed using optimizer estimates.

We compute solution quality similarly to [1, 19], by comparing the total workload cost on an unindexed database $c_{unindexed}$ vs. the total workload cost on the indexes selected by the design tool $c_{indexed}$. We report percent workload speedups according to:

$$\%\ workload\ speedup = 1 - c_{indexed}/c_{unindexed}$$

We compare the quality of the indexes suggested by CoPhy against the greedy algorithm described in Section 4.8 and Appendix 4.8.2, and the FLP-based index selection tool. Since the solution to the problem given by Papadomanolakis et al. is within 0.2% of the optimal solution on *System1* for the pruned candidate set, we present the results from their method and identify them as *FLP*. The scalability of CoPhy is demonstrated by varying the workload cardinality.

4.9.2 Solution Quality Comparison

In this experiment, if D is the size of the database, we allocate xD space to indexes in the final solution. We increase x gradually to observe the improvement in the quality of the selected indexes. For CoPhy, we stop the search process when the solution reaches within 5% of the optimal solution. Figure 4.4 compares the solution quality for TPCH15 using the three INUM-based selection mechanism (Greedy, FLP, and CoPhy) on the commercial system *System1*. On the x-axis, we increase the storage constraint x, and on the y-axis we show the workload speedup after implementing the selected indexes.

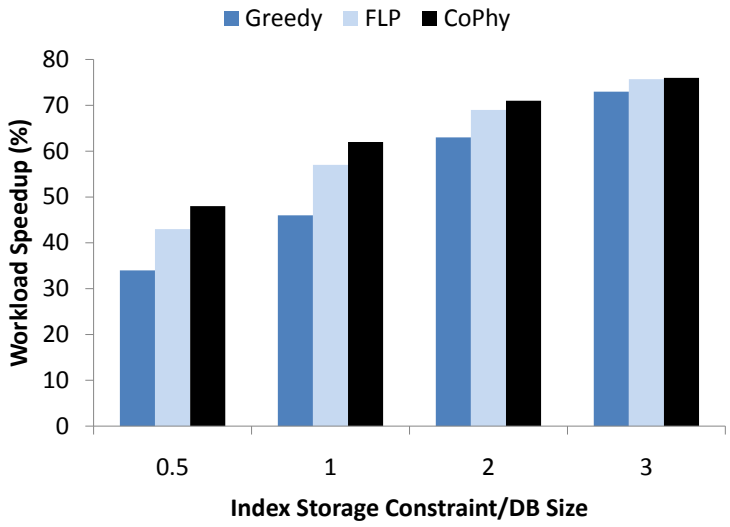


Figure 4.4: Results for index selection tools for TPCH15

Figure 4.4 shows that CoPhy suggests indexes with the highest workload speedup for all values of x on *System1*. As the space allocated for the indexes grows, all tools converge towards selecting indexes with similar workload speedup. CoPhy improves on *Greedy* by fully searching the index combination space, instead of looking at one index at a time. *Greedy* prefers large indexes on tables such as LINEITEM, and misses indexes on the smaller PART and CUSTOMER tables. CoPhy, however, selected a more balanced set of indexes on both large and small tables to achieve

the best solution quality. FLP needs to prune many candidate configurations to keep the problem size under control, hence it misses some plans that CoPhy identifies.

These results demonstrate that CoPhy finds significantly is superior to the greedy algorithm and FLP algorithm running on the same set of candidate indexes. The benefit of CoPhy is more pronounced when less space is available for indexes.

4.9.3 Execution Time Comparison

Figure 4.5 compares the execution time of various index selection algorithms with $x = 1$. On the x-axis, we vary the workload size and on the y-axis we show the execution times for different algorithms. To observe the execution time on larger workloads, we create workloads of 250, 500, 750, and 1000 TPC-H-like queries using TPC-H's *QGen* on TPCH15's query templates, and name them TPCH250, TPCH500, TPCH750, and TPCH1000 respectively. To scale INUM computation for these larger workloads, we use INUM approximations [51]. The approximation method

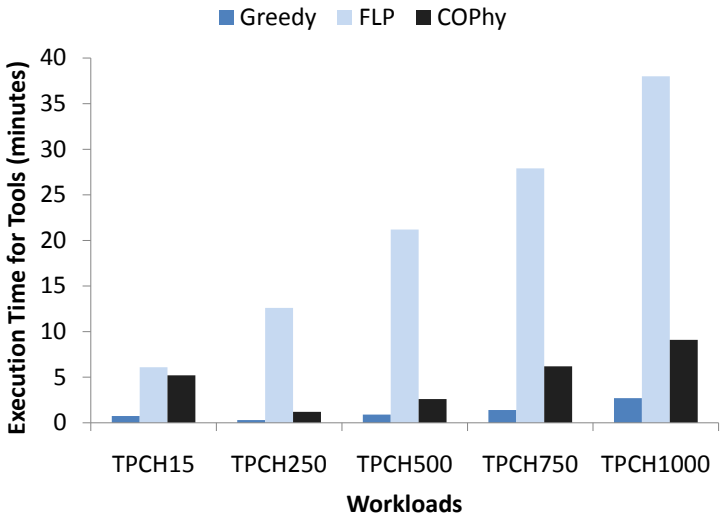


Figure 4.5: Execution time of algorithms for varying query sizes.

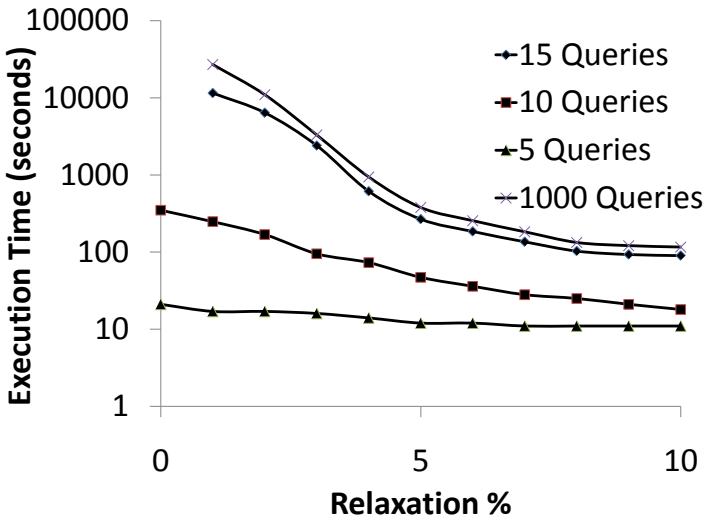


Figure 4.6: Execution time of CoPhy at different relaxations from the optimal

reduces the number of cached plans from 90 to 2 on average for each query, but reduces the accuracy. We run the experiment with index size multiplier $x = 1$, and with 5% quality relaxation for CoPhy. Since the query templates remain the same, the benefit of using the selection tool for these larger workloads remains similar to Figure 4.4, except that the cost approximation for larger workloads reduces the solution quality for both CoPhy and greedy algorithms by about 3%. Note that even though TPCH15 does not use approximations, we compare the running times in the same charts in order to save space. The time taken to build the INUM cache model for the workloads in Figure 4.5 are 24, 2, 4, 9, and 17 minutes respectively. Using workload compression along with INUM's approximation reduces the INUM cache construction overhead to less than one minute for the generated workloads.

We first focus on TPCH15. *Greedy* is the fastest selection tool as it benefits from the index structure described in Appendix 4.8.2 in detail. In this experiment, building the in-memory structures for *Greedy* takes approximately 18 seconds, and running the greedy algorithm takes about 25 seconds. In comparison, the reported execution time for a similar greedy algorithm is about

3.5 hours [53], hence using the index structure speeds up the algorithm by a factor of nearly 300. CoPhy spends about 5.2 minutes to find the solutions, of which about 20 seconds were spent in building the problem from the index variables, about 45 seconds to find the starting greedy solution, and the remaining in solving the problem with LR. FLP spends more than 5 minutes pruning the configurations and building the combinatorial problem. FLP, however, solves the problem faster than CoPhy, since the pruning reduces the problem size and complexity.

Now we consider the generated workloads and observe the scaling behavior of the tools. *Greedy* and FLP scale almost linearly with the problem size. CoPhy's execution time goes up to about 9 minutes for the largest workload, however, about 2.5 minutes are spent in the initializing the greedy solution, hence the LR method scales well with increasing workload cardinality as well.

4.9.4 Quality Relaxation vs. Execution Time

We gradually increase the relaxation distance from the optimal value and observe the improvement in execution time for the LR-based solver. We use *System1* with the index space constraint to be the same as the DB size. We report only the time it takes for the LR algorithm to run, as the greedy algorithm's execution time does not change with the relaxation values. In Figure 4.6, we vary the distance from the optimal value on the x-axis, and on the y-axis we show the execution time of the solver. The three different lines in the figure show the behavior on three subsets of the TPCH15 queries, containing 5, 10, and 15 queries respectively. We also compare against the 1000 TPC-H queries generated in the previous experiment.

Figure 4.6 shows that even for a small workload with 15 queries, achieving the exact optimal value is not feasible. Even after running the solver for more than a day, we could not converge to the optimal solution. When relaxing the solution to be within 1% from the optimal value, the solver converges after a long time. For the 15 query workload, a 5% relaxation provides the best balance between proximity to optimal solution and execution time. When the workload size is reduced

to just 5 queries, CoPhy finds the optimal solution within a minute, this is possible because the problem search space is small enough for the solver to explore every possible combination of indexes. Similarly, for 10 queries, the solver converges to the optimal solution, but takes more time. Consequently, even with small workloads, the ability to relax the targeted solution helps in dramatically reducing the execution time of the solver.

4.9.5 Comparing Against Commercial Tools

Since the commercial designer tools lack a fast cost model like INUM, their runtime is typically much higher than the INUM-based solutions discussed so far. Hence we only show the comparison of the performance of their suggested indexes. We compare CoPhy against two commercial system physical designers implemented on *System1* and *System2* using TPCH15. Since the indexes suggested by the tools on these two systems cannot be directly compared, we normalize the performance on each system by the performance of CoPhy's suggested indexes. If c_{system} is the total workload cost on *System* using the suggested indexes, and c_{cophy} is the workload cost with CoPhy's suggested indexes on *System*, then we report the relative workload speedup: c_{system}/c_{cophy}. On the x-axis we increase the index constraint size as a multiple of the table size, and on the y-axis we report the relative speedup.

Figure 4.7 shows that CoPhy always performs better than *System1*'s designer tool, and performs better than *System2*'s tool when the space allocated for indexes is low. This is a direct result of not pruning the candidate set eagerly. Therefore, we believe that the generic search method of CoPhy helps in improving the query performance of the commercial systems, when compared to the sophisticated and fine tuned greedy methods employed by the commercial tools.

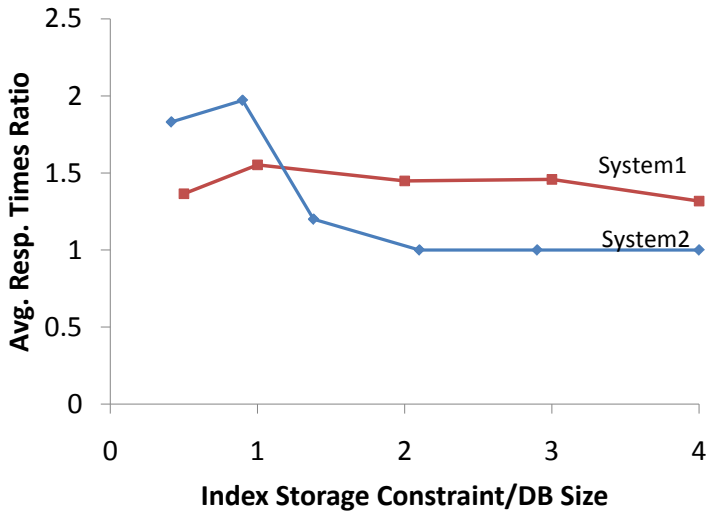


Figure 4.7: Quality comparison against commercial systems

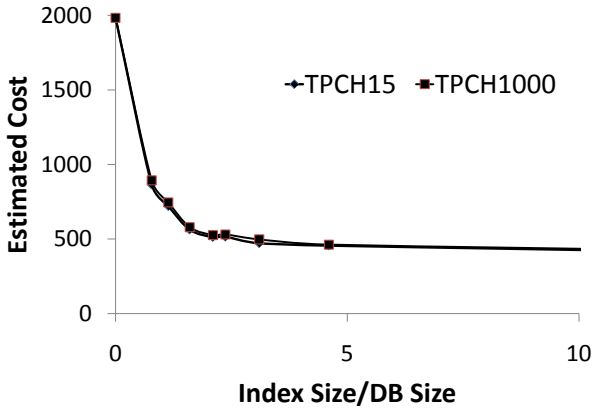


Figure 4.8: Pareto-optimal curve with the storage constraint and the workload costs as soft constraints

Adding Soft Constraints

So far we have discussed index selection problem with hard index storage constraints and a soft constraint on the workload cost being zero. In this experiment, we add more soft constraints to the problem to observe the pareto-optimal surface that CoPhy generates. We replace the hard constraint of the index storage cost with a soft constraint which tries to minimize the storage cost. Let s be the storage cost, and c be the cost of the workload in the new configuration. Using the scalarization technique discussed in Section 4.5.4, CoPhy minimizes the term $\lambda_1 c + \lambda_2 s$. CoPhy begins by considering the extreme points where $\lambda_1 = 0, \lambda_2 \neq 0$ and the other extreme point where $\lambda_1 \neq 0, \lambda_2 = 0$. Then it considers the third point with $\lambda_1 c = \lambda_2 s$, and finally explores other points in between using a binary search-like technique.

Using these two soft constraints, we run the solver with different values of λ_i settings to achieve the pareto-optimal curve, as shown in Figure 4.8. The x-axis shows the storage cost of the suggested indexes, and the y-axis shows the estimated cost of the workload. For each point in the graph, the solver takes approximately 25 seconds and we show 10 different combinations of λ_i values. CoPhy determines the shape of the curve using the first 3 points, hence CoPhy estimates

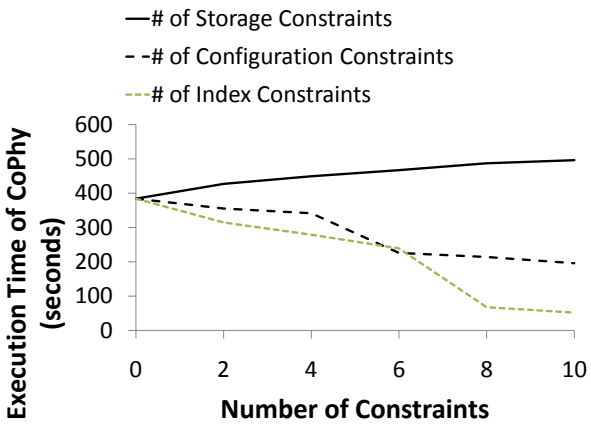


Figure 4.9: Execution time of CoPhy's solver with varying number of constraints.

the shape of the pareto-optimal curve faster than manually changing the hard constraints.

Varying the Constraint Size

In this experiment, we show that increasing the input program size does not affect the scalability of the solver. We investigate the effect of extra constraints by separating them into three categories, i.e., a) index constraints, b) configuration constraints, and c) storage constraints. For the first category, we generate constraints of the form "`For Index I on LINEITEM I not like 'column,%'`". We generate 10 different constraints for 10 columns of the LINEITEM table, and add them to our program. For the second category, we add "`For Query Q in Workload, Cost(Q) < 0.9*EmptyCost(Q)`" to the program for each query Q. For the final category, we add the constraints of the form "the size of the indexes built on a subset of tables must not exceed 90% of the total storage constraint". For example, one of the constraints specifies that LINEITEM's and ORDERS's indexes should not exceed 90% of the total storage space.

Figure 4.9 shows the execution time of the solver (since other times remains constant for all points) for each of the constraint categories, with an increasing number of constraints. On the

x-axis we increase the number of constraints in steps of two, and on y-axis we show the execution time of solver for the modified problems. The presence of configuration constraints in the index selection problem restricts us from using the greedy initialization, hence the execution time of the solver suffers slightly when compared to earlier experiments. As we add the index and configuration constraints, the runtime of the solver is reduced, because adding these constraints to the optimization problem, reduces the size of the search space, thus helping the solver to get solutions faster.

Similarly, adding the configuration constraints reduces the search space by eliminating many configurations that do not provide sufficient cost improvement. For example, with 10 constraints on the configurations, the number of candidate configurations drops from approximately 55900 to about 6300. Although the search space shrinks drastically, the running time of the solver does not go down proportionally, since the majority of the search happens in the reduced space in the normal case as well. The storage constraints, however, add more processing to the solver, as they are harder constraints compared to the other varieties. As more storage constraints are added, the overhead of doing the Lagrangian relaxation increases, consequently the execution time increases albeit almost linearly with a small slope. Note that the storage constraints we add are artificial constraints and represent the worst case behavior for the solver. Whereas, in practice the DBA will more likely add multiple configuration and index constraints, rather than storage constraints.

4.9.6 More Workloads

This section discusses the results for CoPhy on two more workloads. The first one–NREF–is a real-world protein workload. It shows that for simple queries, the quality improvement of CoPhy is on par with the commercial tools. Then we investigate further into the effect of query complexity on the quality of CoPhy's results by using a synthetic benchmark–SYNTH.

Results for NREF

We now discuss the performance of CoPhy and other index selection tools for the NREF workload. The NREF database consists of 6 tables and consumes 1.5 GBs of disk space. The NREF workload consists of 235 queries involving joins between 2 and 3 tables, nested queries and aggregation. Since the queries are simpler and use fewer columns, all selection algorithms converge to the same small set of indexes. In *System1*, these indexes provide 28% workload speedup and on *System2* the speedup is about 23%. *Greedy* outperforms all other algorithms by suggesting indexes in 19 seconds, and *System1* completes in 5.6 minutes. Since we use an approximation method to estimate the query costs, CoPhy uses 470 cached plans for the COP and executes in about 1.4 minutes.

Results for SYNTH

We use the synthetic benchmark to study the behavior of the physical designer in the presence of increasing query complexity. SYNTH is a 1GB star-schema database, containing one large fact table, and 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 ORDERBY clauses. We generate 4 variants of these queries with an increasing number of candidate indexes and interesting orders, hence with increasing complexity. Table 4.1 shows the details of the queries on the database.

Figure 4.10 shows the solution quality of the techniques on various SYNTH workloads when the indexes occupy 25% of the space of the tables. On the x-axis, we increase the query complexity, and on the y-axis we show the solution quality of the techniques on *System1*. When the queries involve only 1 table, all techniques perform equally well. The quality of the greedy solutions, however, reduce as the number of tables, hence the interaction between the indexes become

Workload	SYNTH1	SYNTH2	SYNTH3	SYNTH4
candidates	381	746	3277	4933
cache size	20	110	418	592
# of tables	1	4.6	4.8	10.5

Table 4.1: The details of the synthetic benchmark. The first row lists the candidate index set's size for the workloads, the second one lists the INUM cache sizes, and the last row shows the average number of tables in the query.

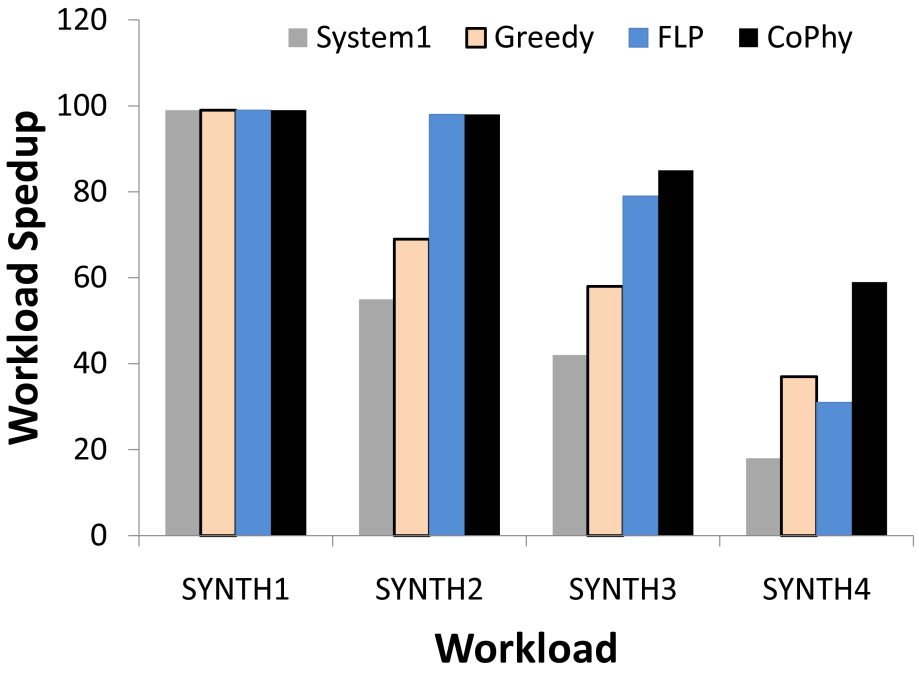


Figure 4.10: Solution quality comparison for SYNTH workloads of increasing complexity.

important, which the greedy mechanism of *System1* and *Greedy* do not address. FLP performs as well as CoPhy when there are only one or two tables in the query. As the number of tables in the

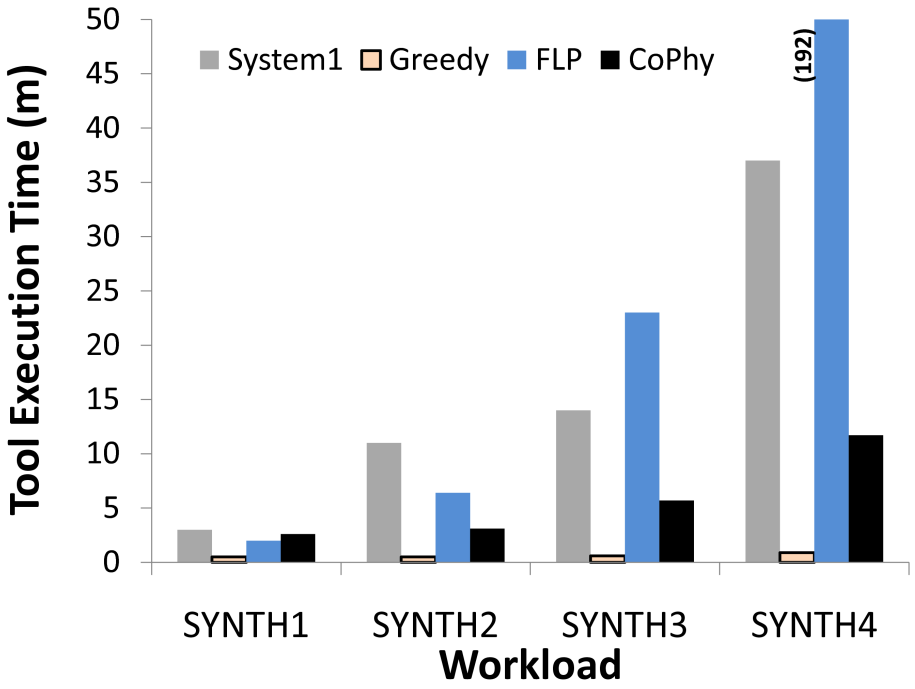


Figure 4.11: Tool execution time comparison for SYNTH workloads of increasing complexity.

table increases, FLP's solution suffers. Without pruning, the FLP formulation creates a COP with about 110 million variables. Solving such a large combinatorial problem is not feasible in today's solvers. Therefore, it prunes away a large fraction of the problem space to produce a problem of the order of tens of thousands. This substantial pruning removes many candidate configurations from the search space, hence reduces the solution quality drastically.

Figure 4.11 shows the tools' execution times for the different workloads. As expected *Greedy* scales the most, since the number of cached plans is relatively small. The FLP technique, scales the worst, as it spends most of its execution time pruning away configurations. *System1* scale well with the complexity of the queries, since it uses the optimizer as a black box, the increase in complexity does not affect its run time. CoPhy scales almost linearly with the INUM cache size, as expected

from the formulation.

4.10 Conclusion

We demonstrate that we can exploit the cost model of INUM to formulate a reasonably sized COP for the physical design problem. We then use the state-of-the-art solvers to solve the COP efficiently. Unlike existing selection tools, this approach is scalable, allow interactive modification of the problem, and most importantly allows DBAs to trade off the execution time against the quality of the suggested solutions. So far in the thesis, we have dealt with offline workloads. In the next chapter, we extend this formulation to online workloads in which the physical designer has to make decisions with no knowledge of the future queries.

Chapter 5

Online Physical Design

The optimization problems discussed so far are all off-line problems. In which the input is given to the optimizer and the optimization solver finds the objective values that satisfy the constraints. While this is a very important problem in practice, in many instances, the workload can change dynamically. The DBA cannot select a set of queries, or a sequence of queries which run in the system. Finding such a set or sequence of queries is especially hard when the user has the power to construct the query by directly writing the SQL or building the query using a query builder. One such example of dynamic query construction is the Sloan Digital Sky Survey (SDSS) [65]. In the SDSS project, the telescopes scan predetermined parts of the sky regularly, and then make them available to the public for querying. Any astronomer, from all over the world, can pose queries to the SDSS server, by using SQL. Since there is no control over the SQL subset that the astronomer needs, the queries typically vary in their use of tables, filters, and columns.

We solve this problem by borrowing from the algorithms community the technique of "Competitive Algorithms". These algorithms assume that an adversary is able to control the input, and the adversary tries to make the algorithm perform as bad as possible by changing the input in response to the decisions taken by the algorithm. The performance of the algorithm is measured

by comparing its performance against an optimal algorithm (OPT) that has complete information about the workload. If the optimal algorithm has a cost of O and the online algorithm has a cost nO, then the online algorithm is considered to be n competitive.

In this chapter we discuss two such competitive algorithms. The first algorithm is suitable for resource constrained environments in which building indexes to speed up the queries is not feasible, and the frequency of the queries is so high that minimal resources need to be used to make the physical design decisions. We therefore optimize the performance of the online workload using only vertical partitions, and provide a competitive algorithm requiring very minimal computation. The second algorithm is more general and is suitable for physical design with indexes as well as other design features. The algorithm has higher overhead than the earlier mentioned one, but we demonstrate that the benefit of using the indexes outweighs the overhead of optimization for real-world workloads.

5.1 Online Algorithm for Vertical Partitions

In this section we describe–OnlinePD– an online physical design algorithm for vertical partitions. `OnlinePD` builds upon algorithms for two classical online problems: (1) the on-line ski rental problem and (2) online algorithm for the metrical task system in which transition costs are symmetrical. The next subsection describes these sub-problems and their known algorithms. The subsection after that describes the algorithms for OnlinePD and proves the bound on its performance.

5.1.1 Related Problems

On-line ski rental is a classical rent-to-buy problem. A skier, who doesn't own skis, needs to decide before every skiing trip that she makes whether she should rent skis for the trip or buy them. If

she decides to buy skis, she will not have to rent for this or any future trips. Unfortunately, she doesn't know how many ski trips she will make in future, if any. This lack of knowledge about the future is a defining characteristic of on-line problems [8]. A well known on-line algorithm for this problem is rent skis as long as the total paid in rental costs does not match or exceed the purchase cost, then buy for the next trip. Irrespective of the number of future trips, the cost incurred by this on-line algorithm is at most twice of the cost incurred by the optimal offline algorithm.

If there were only two configurations and we assume that the cost function $d(\cdot)$ satisfied symmetry, the OnlinePD problem will be nearly identical to on-line ski rental. Staying in the current configuration corresponds to renting skis and transitioning to another configuration corresponds to buying skis. Since the algorithm can start a ski-rental in any of the states, it can be argued that this leads to a 4-competitive algorithm for a given traversal over both configurations. [1]

The above idea has been extended to N configurations by Borodin et al. [9].

$8(N-1)$-competitive algorithm. This algorithm forms the basis for our algorithm in the next sub-section. They consider a complete, undirected graph $G(V,E)$ on $\mathcal{S}$ in which V represents the set of all configurations, E represents the transitions, and the edge weights are the transition costs. By fixing a minimum spanning tree (MST) on G, the algorithm performs an online ski-rental as follows: Pick the maximum weight edge, say (u,v), in the MST and remove it from the MST. This partitions all the configurations into two smaller components and the MST into two smaller trees: MST_1 containing u and MST_2 containing v. Recursively traverse one component until the query execution cost incurred in that component is approximately that of moving to the other component, moving to the other component and traversing it (recursively), returning to the first component (and completing the cycle) and so on. They show that this algorithm is $8(N-1)$-competitive. We extend

[1] A simple 3-competitive algorithm which transitions when the penalty of staying in the current configuration is equal to the *sum* of transition costs, extends the ski-rental idea to include asymmetry in transition costs. We show this in [47]. However, this idea does not extend to N configurations as greedily transitioning to the first configuration that satisfies the penalty leads to an inefficient algorithm.

the above algorithm to non-symmetrical task system.

5.1.2 OnlinePD **Algorithm**

Let the transition cost function d in OnlinePD be represented by a complete directed graph G' We show how to transform this graph into an undirected graph and run Borodin's algorithm on it in such a way that we can bound the competitive ratio for OnlinePD .

In G' replace every pair of directed edges (u,v) and (v,u) with an undirected edge (u,v) and a corresponding transition cost equal to $\sqrt{d(u,v).d(v,u)}$ irrespective of the direction. This transforms G into H. If p is a path in H and p' is the corresponding path in G' (in any one direction), then $\frac{cost(p)}{\sqrt{\rho}} \leq cost(p') \leq \sqrt{\rho} cost(p)$. This allows us to bound the error introduced by using H instead of G.

H does not correspond to an MTS because the triangle inequality constraint is violated. By virtue of using the Borodin's algorithm that constructs an MST, However, `OnlinePD` is resilient to the triangle inequality violation.

Algorithm 2 gives the `OnlinePD` algorithm which uses the Algorithm 3 as a subroutine. We next provide the bound on the algorithm.

Let the maximum rounded weight in the tree F be 2^M. The following proof is inspired by the proof in [9]

Lemma 1 *Any edge in $\mathcal{T}$ of rounded weight 2^m is traversed exactly 2^{M-m} times in each direction.*

Proof We prove by induction on the number of edges in F. For the base case, there are no edges in F, and the lemma is trivially true. For the inductive case, let (u,v) be the maximum weight edge in F used in $\mathsf{traversal}(\cdot)$, and similarly let F_1 and F_2 be the trees obtained by removing (u,v). Now the edge (u,v) is traversed exactly once in each direction as required by the lemma. By the

Input: Directed Graph: $G(V, E_o)$ with weights corresponding to $d(\cdot)$, Query Sequence: σ

Output: Vertex Sequence to process σ: $u_0, u_1, \ldots$

Transform G to undirected graph $H(V,E)$ s.t. $\forall (u,v) \in E$ weight $d_H(u,v) \leftarrow \sqrt{d(u,v) \cdot d(v,u)}$;

Let $B(V,E)$ be the graph H modified s.t. $\forall (u,v) \in E$ weight $d_B(u,v) \leftarrow d_H(u,v)$ rounded to next highest power of 2;

Let F be a minimum spanning tree on B;

$\mathcal{T} \leftarrow \mathsf{traversal}(F)$;

$u \leftarrow S_0$;

while *there is a query q to process* **do**

  $c \leftarrow q(u)$;

  Let v be the node after u in $\mathcal{T}$;

  while $c \geq d_B(u,v)$ **do**

    $c \leftarrow c - d_B(u,v)$;

    $u \leftarrow v$;

    $v \leftarrow$ the node after v in $\mathcal{T}$;

  end

  Process q in u;

end

Algorithm 2: OnlinePD(G)

inductive hypothesis, each edge of F_1 of rounded weight 2^m is traversed exactly 2^{M_1-m} times in each direction in the traversal $\mathcal{T}_1$, in which M_1 is the maximum rounded weight in F_1. Since $\mathcal{T}$ includes exactly 2^{M-M_1} traversals of $\mathcal{T}_1$, it follows that each such edge is traversed 2^{M-m} is each direction in $\mathcal{T}$. Exactly the same reasoning applies for edges in F_2.

Theorem 1 *Algorithm* OnlinePD *is* $4(N-1)(\rho + \sqrt{\rho})$*-competitive for N configurations and asym-*

Input: Tree: $F(V,E)$

Output: Traversal for F: $\mathcal{T}$

if $E = \{\}$ **then**
  $\mathcal{T} \leftarrow \{\}$;
else if $E = \{(u,v)\}$ **then**
  Return $\mathcal{T}$: Start at u, traverse to v, traverse back to u;
else
  Let (u,v) be a maximum weight edge in E, with weight 2^M;
  On removing (u,v) let the resulting trees be $F_1(V_1,E_1)$ and $F_2(V_2,E_2)$, where $u \in V_1$, and $v \in V_2$;
  Let maximum weight edges in E_1 and E_2 have weights 2^{M_1} and 2^{M_2} respectively;
  $\mathcal{T}_1 \leftarrow \mathsf{traversal}(F_1)$;
  $\mathcal{T}_2 \leftarrow \mathsf{traversal}(F_2)$;
  Return $\mathcal{T}$: Start at u, follow $\mathcal{T}_1$ 2^{M-M_1} times, traverse (u,v), follow $\mathcal{T}_2$ 2^{M-M_2} times, traverse (v,u);
end

Algorithm 3: $\mathsf{traversal}(F)$

metry constant ρ.

Proof We shall prove that during each traversal of F, the following two statements are true: (i) the cost (ii) the cost of the offline optimal is at cost of OPDA during any single traversal is constant with respect to the length of σ.

To prove (i), recall from Lemma 1 that any edge in $\mathcal{T}$ of rounded weight 2^m is traversed exactly 2^{M-m} times in each direction. Thus the total rounded weight traversed for an edge is $2 \cdot 2^{M-m} \cdot 2^m = 2 \cdot 2^M$. By construction of the algorithm the total processing cost incurred during $\mathcal{T}$ at a node just before a traversal of this edge is $2 \cdot 2^M$. The total transition cost incurred during $\mathcal{T}$

in a traversal of this edge is at most $2 \cdot 2^M \sqrt{\rho}$, since the cost $d(\cdot)$ can be at most $\sqrt{\rho}$ times larger than the corresponding $d_B(\cdot)$. This proves (i) as there are exactly $N-1$ such edges.

We prove (ii) by induction on the number of edges in F. Suppose F has at least one edge, and (u,v), F_1, and F_2 are as defined in $\mathsf{traversal}(\cdot)$. If during a cycle of $\mathcal{T}$, OPT moves from some vertex in F_1 to some vertex in F_2, then since F is a minimum spanning tree, there is no path connecting F_1 to F_2 with a total weight smaller than $d_B(u,v)/(2\sqrt{\rho}) = 2^{M-1}/\sqrt{\rho}$. Otherwise during the cycle of $\mathcal{T}$, OPT only stays in one of F_1 or F_2; w.l.o.g. assume F_1. If F_1 consists of just one node u, and OPT stays there throughout the cycle of $\mathcal{T}$, then by definition of the algorithm, OPT incurs a cost of at least $d_B(u,v) = 2^M \geq 2^{M-1}/\sqrt{\rho}$. If F_1 consists of more than one node, then by the induction hypothesis, OPT incurs a cost of at least $2^{M_1-1}/\sqrt{\rho}$ per cycle of T_1. Since during one cycle of $\mathcal{T}$ there are 2^{M-M_1} cycles of $\mathcal{T}_1$, OPT incurs a cost of at least $2^{M-1}/\sqrt{\rho}$. This completes the proof.

We next describe the cost estimation modules.

5.1.3 Transition Cost Model

Given N configurations, to determine the optimal target configuration, OnlinePD needs to compute N^2 transition costs. Naively computing the transition cost by actually building each configuration requires reorganizing at least hundreds of gigabytes of data on disk. The cost of doing so is prohibitively high, and can take several hours to complete.

Therefore, we build a transition cost model to efficiently estimate the cost of such transitions without actually building the target configurations. Specifically, we model the cost of transitioning to a new configuration using the standard `BulkCopy` tool.

First, we describe our rationale behind selecting the `BulkCopy` tool. We considered two alternative methods to transition to new configurations: `BulkUpdate`, and `BulkCopy`. To understand the difference between these two options, consider a configuration S_1 which partitions

table `T1` into two partitions $\texttt{T1}_1\texttt{(a,b,c)}$ and $\texttt{T1}_2\texttt{(a,d)}$, column `a` being the primary key for `T1`. We need to transition into configuration S_2 with a single table `T1(a,b,c,d)`.

In the `BulkUpdate` approach, we update the schema for $\texttt{T1}_1$ by allocating space for column `d` and then use SQL *update* statements to copy the values of `d` from $\texttt{T1}_2$ to $\texttt{T1}_1$ before dropping $\texttt{T1}_2$. In the `BulkCopy` approach, we first export the data from T1_1 and T1_2 into flat files before deleting the partitions and then copy the data back to the table `T1`.

In our experiments, `BulkCopy` was nearly 7 times faster than `BulkUpdate`. The reasons behind this performance difference are: 1) `BulkCopy` exploits sequential I/O when copying data into a new table, 2) `BulkCopy` in native format incurs no type casting overhead, and 3) `BulkCopy` avoids `BulkUpdate`'s transaction logging overhead. `BulkCopy` also allows us more control over space allocation. For example, the database may delay deallocation of space when dropping columns from existing tables, adversely impacting query performance.

To model the I/O cost of `BulkCopy`, we make the following two observations:

1. Copying data into a database is about 10 times more expensive than exporting the data out of a database. Hence, we focus on the import cost in our model.

2. The cost of importing scales linearly with the amount of data being copied into the database, due to sequential I/O.

Given these observations, we can model the cost of importing a partition P, with R_P rows and average column width W_P as:

$$BCP(P) = cR_PW_P + kR_P \tag{5.1}$$

Where c is the cost per byte of copying data into the database and k is the cost per row of constructing the clustered primary key index. These constants are database dependent and can be easily determined using regression on a few "sample" `BulkCopy` operations.

We use Equation 5.1 to model the cost of moving from a configuration S_i to another configuration S_j. Let configuration S_i consist of partitions $\{T1_i,...,Tm_i\}$ and let S_j consist of partitions $\{T1_j,...,Tn_j\}$. Let Δ_{ij} be the set difference $\{T1_j,...,Tn_j\}$ - $\{T1_i,...,Tm_i\}$, which denote all tables in S_j but not in S_i. The cost of transitioning from S_i to S_j can be computed as:

$$d(S_i, S_j) = \sum_{t \in \Delta_{ij}} BCP(t) \tag{5.2}$$

Here we assume that the cost of dropping the partitions in S_i is zero. We experimentally verified that Equation 5.2 estimates the cost of transitions with 87% accuracy.

Finally, using Equation 5.2, we show that the asymmetry constant ρ is bounded by a constant factor. By definition $\rho = \max_{ij} d(S_i, S_j)/d(S_j, S_i)$. The ratio is maximized when transitioning between a configuration S, with all columns of a table grouped together, to a configuration S' with each column in a separate partition. For simplicity, let the table consist of r rows and contains y columns, with each column sized at w bytes. Then

$$d(S', S) = crwy + kr \;\; \& \;\; d(S, S') = y(crw + kr)$$
$$\Rightarrow \rho \leq d(S', S)/d(S, S') = \frac{y(cw + k)}{ycw + k}$$

5.1.4 Query Cost Estimation - QCE

In OnlinePD for every query we need to compute its cost against all N neighboring configurations. The brute-force method is to evaluate the cost of the query against all possible configurations. For a workload with m queries and N target configurations, this would lead to estimating mN query-configuration pairs. The traditional approach of asking the optimizer for the cost of querying on each configuration is well-known to be very expensive (Chapter 3). The query optimization

overhead for each configuration largely contributes to the inefficiency and long running time of automated physical design tools.

We make an important observation in Chapter 3: the plan of a query across several configurations is invariant. Thus, instead of optimizing each query against every configuration, we can estimate query costs by reusing optimized plans from earlier invocations of the optimizer. The optimal plan is cached with very little space overhead. This observation has been explored when configurations correspond to set of indices on tables. However, the table itself is unpartitioned. In this section, we show that the same idea with a twist works even for configurations corresponding to vertical partitions of a table. We describe our idea through an example.

Example of Plan Re-use: Consider two tables `T1(a,b,c,d)` and `T2(e,f,g,h)` with primary and join keys `a` and `e`, respectively. Let the tables in configuration S_1 be vertically partitioned into $\texttt{T1}_1\texttt{(a,b,c)}$, $\texttt{T1}_2\texttt{(a,d)}$, and `T2(e,f,g,h)`. Consider the following query q on the two tables:

```
q = select T1.b, T2.f from T1,
T2 where T1.a =  T2.e and T1.c
> 10 and T1.d = 0
```

The query is optimized to have the plan shown in Figure 5.1, and having cost $q(S_1)$ in configuration S_1. Now suppose we have to evaluate the cost of query in two new configurations: S_2, defined as $\texttt{T1}_1\texttt{(a,c)}$, $\texttt{T1}_2\texttt{(a,b,d)}$, and `T2(e,f,g,h)`, in which column `T1.b` moves from one partition to the other; and S_3 defined as $\texttt{T1}_1\texttt{(a,c,d)}$, $\texttt{T1}_2\texttt{(a,b)}$, in which column `T1.d` moves from one partition to the other. Figure 5.1 also shows the optimal plans for S_2 and S_3. Observe that the join order and join method of the plans remains exactly the same in S_2, while it changes completely in S_3. This happens because the selection of join order and join methods depends heavily on (1) the size of the filtered partitions, and (2) the available ordering on the partitions. In the example, the distribution of columns on which individual partitions of `T1` are filtered (i.e. `T1.c` and `T1.d`) does not change between S_1 and S_2, the size of the filtered partition remain

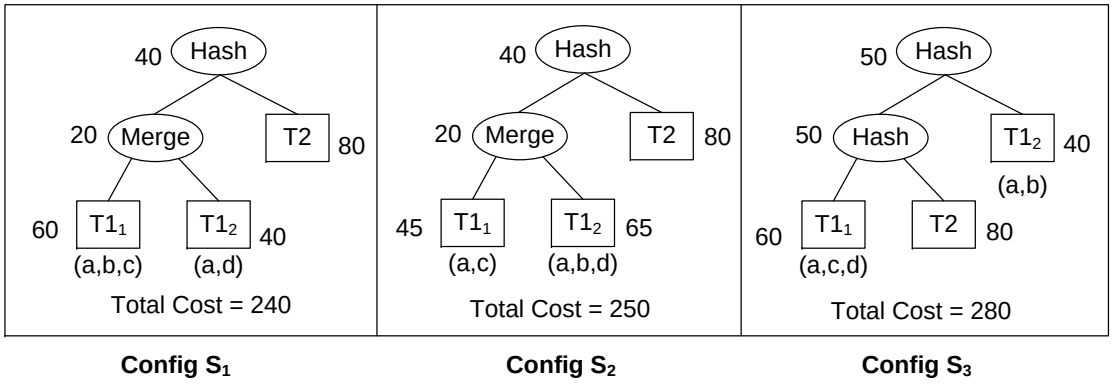


Figure 5.1: Plan graph comparison for configurations with only vertical partition changes. The boxes represent operations, and the numbers next to them show the estimated costs for those operations. The plans remain the same for configurations S_1 and S_2, as they have the same distribution of *filtering* columns, i.e., c and d. Configuration S_3's query plan is significantly different, since it has a different distribution of the *filtering* column.

the same. Since the partitions are always indexed on their primary keys, there is no difference in join ordering. The join method and orders, however, changes in S_3, as one of the columns involved in WHERE-clause of the query has moved from one partition to other.

To formalize, the plan for a configuration S_i can be correctly used for S_j if the following conditions are satisfied:

1. The number of partitions for each table does not change.

2. The distribution of filtering columns on the above partitions does not change.

3. The page size distribution of the partitions does not change *drastically*.

Condition 1 guarantees the same number of joins in the plan for both configurations. For example, if S_i partitions a table `T1` into two partitions, and S_j partitions it into three partitions, then queries on S_i only join once to reconstruct the rows of the original table, while queries on S_j have to join twice.

Condition 2 guarantees the same intermediate join results between S_i and S_j so that the optimizer selects the same join order and join method to find the optimal plan.

To understand the need for Condition 3, consider the following example scenario. Let $\texttt{T3}(c_1, c_2, \ldots, c_{100})$ be a large table with 100 columns of equal width, and c_1 and c_{100} are the filtering columns. Assuming that S_i splits the table `T3` as $\texttt{T3}_1(c_1)$, $\texttt{T3}_2(c_2, \ldots, c_{100})$ and S_j splits the table as $\texttt{T3}'_1(c_1, \ldots, c_{50})$, $\texttt{T3}'_2(c_5 1, \ldots, c_{100})$. The page distribution for T3's partitions in S_i is highly skewed, while the one in S_j is uniform. Typically, for S_j, the optimizer joins $\texttt{T3}'_1$ with $\texttt{T3}'_2$ first, before joining the result with other tables. For S_i, however, the optimizer takes advantage of the small page size for $\texttt{T3}_1$ to delay joining with $\texttt{T3}_2$ as long as possible. Hence reusing the plan for S_j in S_i will provide inaccurate results. [2]

[2]In our online algorithms, we typically estimate costs for many configurations which differ from each other by

If the above three conditions are satisfied, we prove by contradiction that the plans will use the same join order and join methods. Suppose the join method and join order for S_i and S_j are J_1 and J_2 respectively. By our assumption, J_1 and J_2 are different.

Without loss of generality, suppose J_1 costs less than J_2 if we ignore the costs of scanning the partitions. Since the configurations S_i and S_j have the same ordering (primary key order for all partitions), and they select the same number of rows from the partitions, and the page size of the filtered rows are similar, then J_1 can still be used for S_j. Since using J_1 reduces the total cost for running the query on S_j, it implies that J_2 must not have been the optimal plan, contradicting our assumption that J_2 was part of the optimal plan.

Since optimizers spend most of their time considering alternate join orders and join methods, we can save a significant amount of computation while estimating the cost of S_j. If we already know the optimal plan for S_j, we can memoize it for future use.

The Plan-Cache: An important issue in implementing the cache-and-reuse principle is "What is cached?". We cache the join methods and join orders, and replace the partition scans with place holders. This stripped plan structure is cached using a unique key representing the query and the configuration's filtering column distribution among partitions. Revisiting the example of plan reuse in Figure 5.1, we cache the structure shown in Figure 5.2 with the key $[query-id, T1((c),(d))]$ from the plan returned by the optimizer in S_1. The $query-id$ component guarantees that there are no collisions between plans of different queries, and the second component $T1((c),(d))$ encodes the distribution of the filtering columns into two partitions containing (c) and (d) respectively.

Cost Estimation Procedure: To estimate the query cost for a new configuration, we first build the key using the query's identity and the filtering column distribution of the configuration. We

just one attribute. Since only one attribute differs between two configurations, the page size distribution between configurations remain similar. We intend to study this aspect in our future research to identify the maximum distance between the page size distributions that allows plan reuse.

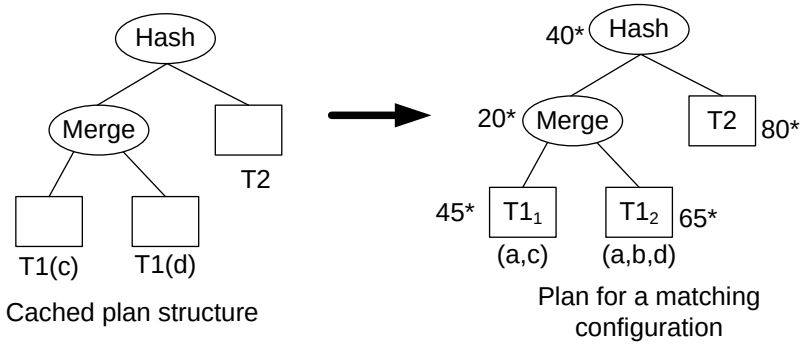


Figure 5.2: The structure cached with the key $[query-id, T1((c),(d))]$. Note that all scanning nodes are place holders for the actual partitions and the costs are missing. The plan on right is the estimated plan for the new configuration S_2, with the partitions and estimated costs filled in.

look up the cached plan structure using this key. The cached structure gives us the join orders and join methods, but does not provide the full query cost. To estimate the query cost, we have to estimate the partition scanning costs and fill them in to the place-holders. We also need to estimate the costs of joining. Now we describe the techniques to estimate these costs.

For full index scans, the cost of scanning the index is proportional to the size of the index, since the DBMS has to scan all pages of the index sequentially. If s is the estimated size of the primary key index, then the cost of scanning is αs, where α is a proportionality constant specific to the database. For clustered index scans, we determine the fraction of the index scanned using the cached plan. Let c be the IO cost of the scanning operation in the cached plan, and s_0 is the size of the index scanned in the cached plan. Then we compute the scanned fraction $f = c/(\alpha s_0)$, as αs_0 be the cost of scanning the entire index. The cost of a cluster scan on the index with size s in the new configuration is $\alpha f s$. The proportionality factor α is easily precomputed using a few "sample" partition scans.

To estimate the cost for joining partitions using the join methods from the cache, we adopt System R's model, developed by Selinger et al. [66]. The System-R cost model gives us an upper bound on the actual join costs, and according to our experiments predicts the plan cost with 1.3%

accuracy on average (Section 5.1.7).

To summarize, we first generate a key for the current combination of configuration and query, to check against plan cache. If the plan is present, then we reuse the join order and join methods to estimate the total cost of the plan. If the plan is not yet present in the cache, we call the optimizer and cache the resulting plan. We show in Section 5.1.7 that using QCE for query cost estimation reduces the cost estimation overhead significantly.

5.1.5 Candidate Configurations

We next describe a workload-based heuristic for determining the set of *N* configurations evaluated in OPT. An ad hoc partitioning of tables leads to an exponential number of configurations. Fortunately, astronomy workloads are template-based and can be summarized compactly through query prototypes [73]. (Workload evolution still occurs as the set of templates change over time). A query prototype is defined as the set of attributes a query accesses such that queries with identical prototypes make up an equivalence class in the workload. For instance, the following queries from an astronomy workload [65] form the same prototype:

SELECT objID, ra, dec FROM PhotoPrimary WHERE dec between 2.25 and 2.75

SELECT top 1 ra, dec FROM PhotoPrimary WHERE objID = 5877311875315

Candidate configurations are generated as new query prototypes in the workload sequence. Given a table `T1(a,b,c,d,e)` (`a` is the primary key) and a prototype `P1 = (a,c)`, we generate a candidate configuration for `P1` corresponding to tables `T2(a,c)` and `T3(a,b,d,e)`. Thus, each candidate is optimized for the scan cost of a specific class of queries. We also merge non-overlapping prototypes to generate new candidates. For example, given prototype `P2 = (a,b)`, we generate a candidate from both `P1` and `P2` consisting of tables `T4(a,c)`, `T5(a,b)`, and

`T6(a,d,e)`. The intuition comes from enumeration-based, offline vertical partitioning algorithms [52][34] in which candidates are enumerated by coalescing groups of columns in existing configurations in a pairwise manner. The resulting candidates gradually reduce the expected total query execution cost for the workload. We plan to study the problem of enumerating N (the size of the configuration space) in more detail in future works.

5.1.6 Architecture

The online physical design tool is co-located with the proxy cache of the SDSS database (Figure5.3). The proxy cache receives a sequence of queries. Ideally, given a new query, the caching and the configuration decisions will be simultaneous. However, we implement it by first sending the query to the OPT module, which, along with the Cost Estimation module and the Configuration Manager, determines if a transition is required. The query is then presented to the cache for execution. The cache may determine that new objects need to be loaded for executing the current query. If data objects are loaded, the specification of the new objects is sent to the Configuration Manager. The Configuration Manager builds up the new set of possible and relevant configurations for the OnlinePD module to consider. The Configuration Manager instructs the cache to perform the lowest-cost cache transition and provides the starting configuration for OnlinePD .

5.1.7 Experiments

We implemented our online-partitioning algorithms and cost estimation techniques in the proxy cache of the Sloan Digital Sky Survey (SDSS) [65], an astronomy database. The proxy cache reside in the mediation middle-ware of an astronomy federation in which SDSS participates. Performance of these caches is critical to support access to large volumes of scientific data by multiple users. Our goal is to improve the I/O performance in the proxy cache through automated, online tools. We

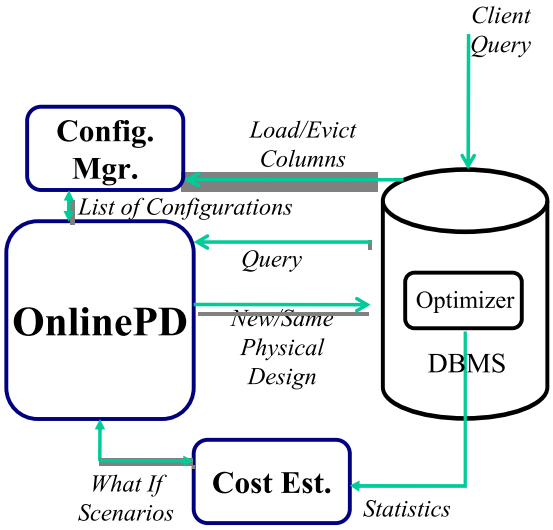


Figure 5.3: Architecture for online physical design tool.

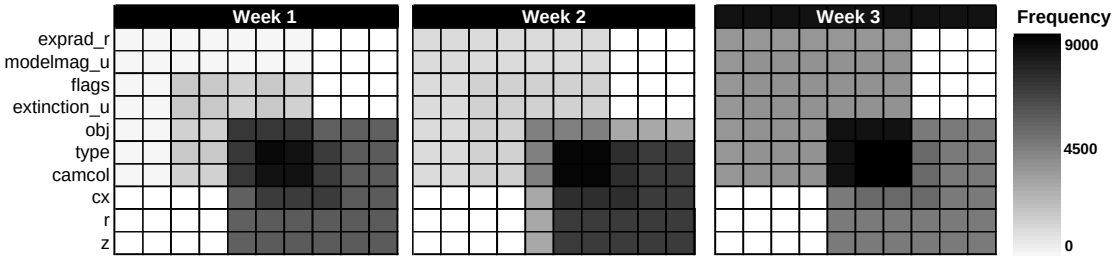


Figure 5.4: Affinity matrix (co-access frequency) for ten selected attributes from the *PhotoObjAll* table.

describe the experimental setup in detail before presenting our main results. This includes analysis of the workload evolution over time, the performance of various online and off-line algorithms, and the accuracy of our cost estimation modules.

Experimental Setup

Workload Characteristics: We used a month-long trace from SDSS consisting of 1.4 million read-only queries. These consist of both simple queries on single tables and complex queries joining multiple tables. Despite a large number of queries, the workload is defined by a small number of query prototypes For instance, the 1.4 million queries in the trace are characterized by as few as 1176 prototypes. However, fewer prototypes does not indicate a lack of workload *evolution*. On the contrary, there is considerable *evolution* in the workload in that new prototypes are introduced continually and prior prototypes may disappear entirely from subsequent queries in the workload sequence.

Figure 5.4 captures workload evolution for the first three weeks of the trace. It shows the affinity matrix for ten attributes from a single table in which each grid entry corresponds to the frequency with which a pair of attributes are accessed together (ordering of attributes are the same along the row and column). The basic premise is that columns that occur together and have similar frequencies should be grouped together in the same relation [49]. The results show that column

groupings change on a weekly basis. An online physical design tool which continuously monitors the workload can evaluate whether transitioning to a new configuration will lead to an improvement in overall cost.

Comparison Methods: We now contrast the performance of OnlinePD with several other algorithms. OnlinePD has polynomial-time complexity and finds the minimal spanning tree using Prim's algorithm. It is a general algorithm that makes no assumptions about the workload. However, this generality comes at a cost. Namely, given some knowledge about the characteristics of a specific workload, we can design highly-tuned heuristic algorithms. To measure the cost of generality, we compare OnlinePD with `HeuPD` [47], a greedy heuristic algorithm designed specifically for SDSS workloads. OnlinePD is a natural progression of `HeuPD`, which we presented earlier [47] as a workload-dependent algorithm that is specifically tuned for SDSS. `HeuPD` transitions immediately to the next best configuration by tracking the cumulative penalty of remaining in the current configuration relative to every candidate for each incoming query. A transition is made once `HeuPD` observes that the benefit of a new configurations exceeds a threshold, which is defined as the square root of the product of the bi-directional transition costs between the current and new configurations.

We also consider `AutoPart`, an existing, offline vertical partitioning algorithm. `AutoPart` is presented with the entire workload as input during initialization. This incurs an initial overhead to produce a physical design layout for the workload, but it can service the workload with no further tuning. Since `AutoPart` is not adaptive, it does not produce the optimal physical layout. `AutoPartPD` is another physical design strategy that employs the offline `AutoPart` algorithm. It is online in that `AutoPart` is rerun on a daily basis, and it is prescient in that the workload for each day is provided as input to the algorithm *a priori*. This provides a lower query response time compared with `AutoPart`, in exchange for increased transition cost. `NoPart` serves as the base case, in which no vertical partitioning is used on the tables.

Performance Criteria: We measure the cost of algorithms in terms of average response time of queries executed in SDSS. This is the measure from the time a query is submitted until the results are returned. If a transition to a new configuration is necessary, the algorithm undergoes a transition before executing the query. This increases the response time of the current query but amortizes the benefit over future queries. Our results reflect average response time over the entire workload.

Database: For I/O experiments, we executed queries against a five percent sample of the DR4 database (roughly 100GB in size). Although sampling the database is less than ideal, it is necessary to finish I/O experiments in a reasonable time for real workloads. Given the time constraints, we compromised database size in order to accommodate a larger workload, which captures workload evolution over a longer period. To sample the database, we first sampled the fact table consisting of all celestial objects (*PhotoObjAll*) and then preserved the rows in the dimension table containing the foreign keys required by the fact table.

The data is stored in a commercial DBMS (*System1*) on a 3GHz Pentium IV workstation with 2GB of main memory and two SATA disks (a separate disk is assigned for logging to ensure sequential I/O). *System1* does not allow for queries that join on more than 255 physical tables. This is required in extreme cases in which the algorithm partitions each column in a logical relation into separate tables. Hammer and Namir [34] show that between the two configurations in which each column is stored separately or all columns are stored together, the preferred configuration is always the latter. In practice, this configuration does not arise because the cost of joining across 255 tables is so prohibitive that our algorithm never selects this configuration. Finally, to reduce the large number of configurations, we do not partition tables than are less than 5% of the database size. This leads to a large reduction in the number of configurations, with negligible impact on I/O performance benefit. Some other heuristics for reducing N are described in the technical report [48]. The total number of configurations were about 5000.

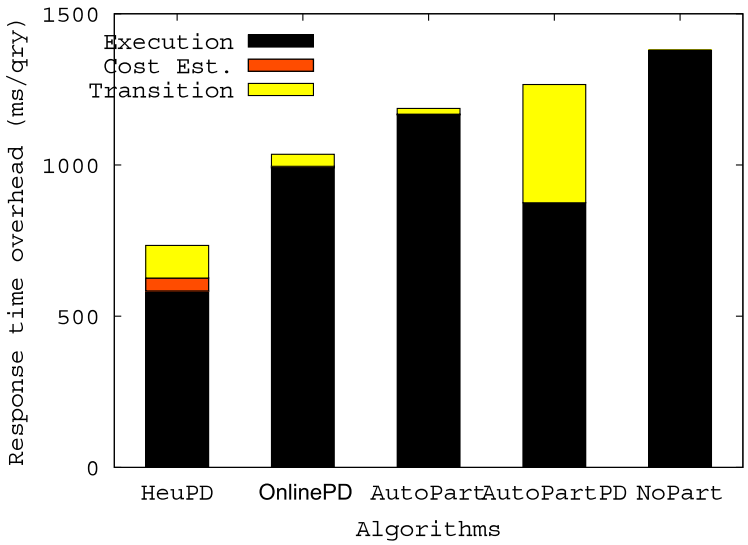


Figure 5.5: Distribution of response time overhead.

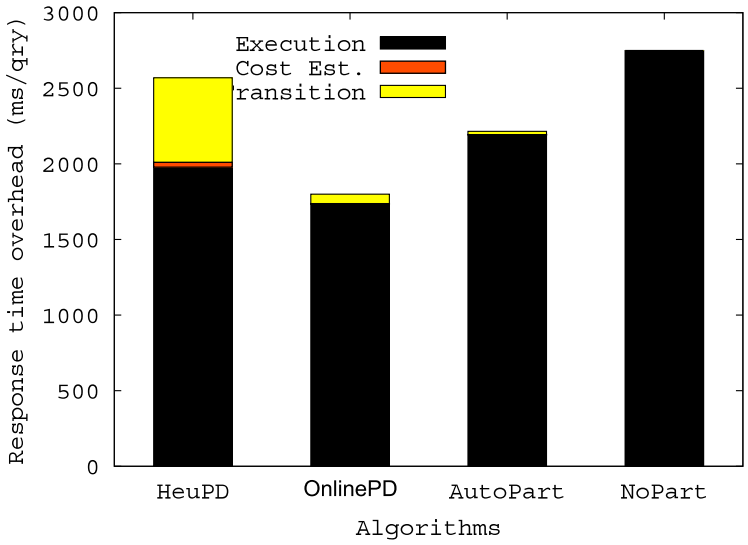


Figure 5.6: Response time overhead for an adversarial workload.

Results

We measured I/O performance by measuring the query response time on the proxy cache using the sampled database. Figure 5.5 provides the division of response time among query execution, cost estimation using the optimizer, and transitions between configurations. (The total response time is

averaged over all queries). It is not surprising that HeuPD, which is tuned specifically for SDSS workloads, performs best with an average query execution time of 583 ms and a two times speedup over NoPart overall. However, OnlinePD increases the cost by just a factor of 1.7. This overhead is low considering that OnlinePD is general and makes no assumptions regarding workload access patterns. In fact it improves on the performance of NoPart by a factor of 1.5. NoPart suffers due to higher scan costs associated with reading extraneous columns from disk. Likewise, AutoPart suffers by treating the entire workload as an unordered set of queries and providing a single, static configuration during initialization. Surprisingly, even AutoPartPD did not improve response time beyond the offline solution because the benefits of periodic physical design tuning is offset by a higher transition cost. Thus, adapting offline solutions to evolving workloads is challenging because they do not continuously monitor for workload changes nor account for transition cost in tuning decisions.

Another interesting feature of the results is that OnlinePD incurs much lower transition costs than HeuPD. This artifact is due to the pessimistic nature of OnlinePD . It transitions only if it sees significant advantages in the alternative configuration. On the other hand, HeuPD responds quicker to workload changes by transitioning to any candidate configuration perceived to benefit I/O performance for the immediate sequence of queries. This optimism of HeuPD is tolerable in this workload but can account for significant transition costs in workloads that change faster than the SDSS workload. To appreciate the generality of OnlinePD over a heuristic solution, we evaluated a synthetic SDSS workload that is adverserial with respect to HeuPD (Figure 5.6). In particular, the workload is volatile and exhibits no stable state in the access pattern, which causes HeuPD to make frequent, non-beneficial transitions. As a result, Figure 5.6 shows that OnlinePD exhibits a lower query execution time and a factor of 1.4 improvement overall when compared with HeuPD.

Figure 5.5 also shows the average response time of performing cost estimation (time spent

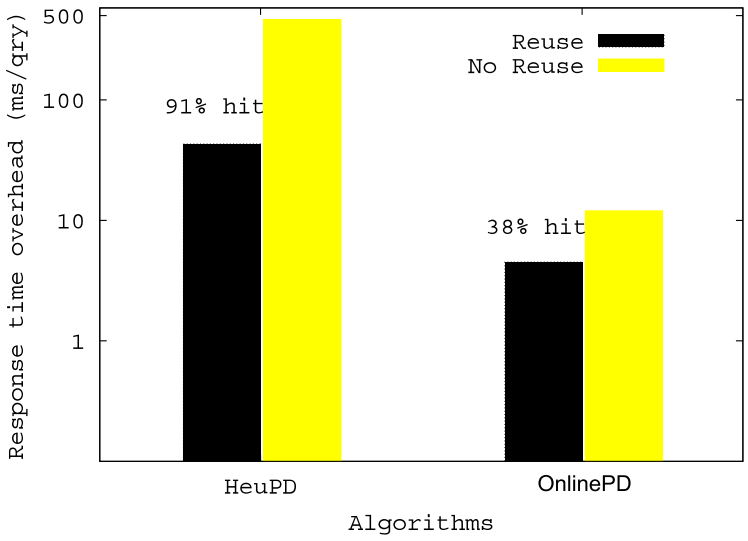


Figure 5.7: Efficacy of query plan reuse in cost estimation.

querying the optimizer). For AutoPart, this is a one-time cost incurred during initialization. In contrast, cost estimation is an incremental overhead in OnlinePD and HeuPD. HeuPD incurs a ten folds overhead in cost estimation over OnlinePD (43 ms versus 4 ms). This is because HeuPD incurs 93 calls to the optimizer per query. Thus HeuPD is a very slow algorithm. This make OnlinePD very attractive for proxy caches that receive a stream of queries in which decisions have to be made fast.

Figure 5.7 illustrates the importance of query plan reuse (note the log scale). Reuse is far more useful in HeuPD by providing a ten-fold reduction in cost estimation (from 465 ms to 43 ms) and avoiding 91% of the calls to the optimizer. OnlinePD also benefits significantly by QCE, which exhibits a three times reduction in cost estimation overhead. In fact, without plan reuse, the total average response time of HeuPD is 1150 ms, which would lag the total response time of OnlinePD by 4 ms. Thus, QCE offsets a majority of the overhead of cost estimation.

Finally, we examine the average transition cost in Figure 5.5. AutoPart only incurs transition costs during initialization, while NoPart incurs no transition cost. AutoPartPD incurs the highest overhead, requiring a complete reorganization of the database on a daily basis. HeuPD

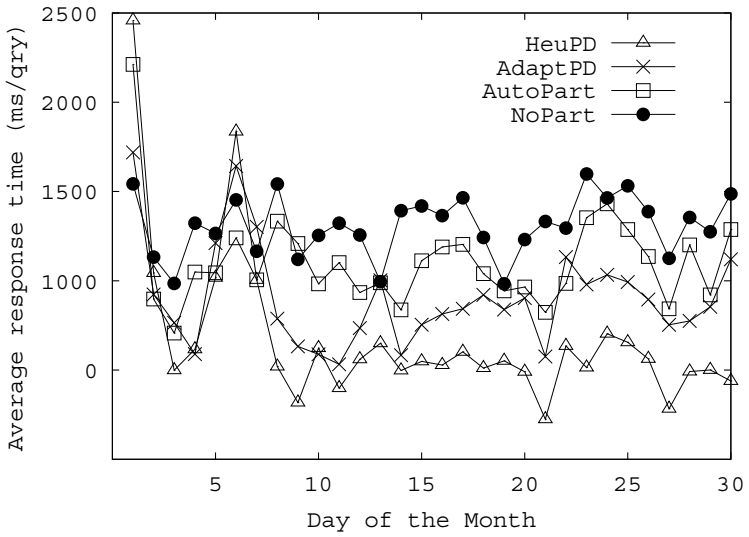


Figure 5.8: Response time overhead on a daily basis.

makes 768 minor configuration changes, compared with 92 for OnlinePD which leads to a three times overhead in transition cost (113 ms compared with 43 ms). This illustrates that `HeuPD` is quicker at detecting and adapting to changes in the workload compared with OnlinePD because it is specifically tuned for SDSS.

Figure 5.8 charts the average response time (both query execution and transition cost) on a daily basis for various physical design strategies.

Figure 5.9 shows the cumulative distribution function (CDF) of the error in estimating costs of queries using QCE instead of calling the optimizer directly. The error in cost estimation is determined by computing

$$abs\left(1-\left(\frac{\texttt{QCE est. query cost}}{\texttt{Optimizer est. query cost}}\right)\right) \tag{5.3}$$

Consider the dashed line in the plot, which corresponds to the errors in cost estimation for all queries. Although the average cost estimation error is only 1.3%, the plot shows that the maximum error in cost estimation is about 46%, with about 14% of the estimations with more than 10% error. Inspecting high error estimations reveals that the errors occur in queries with estimated costs

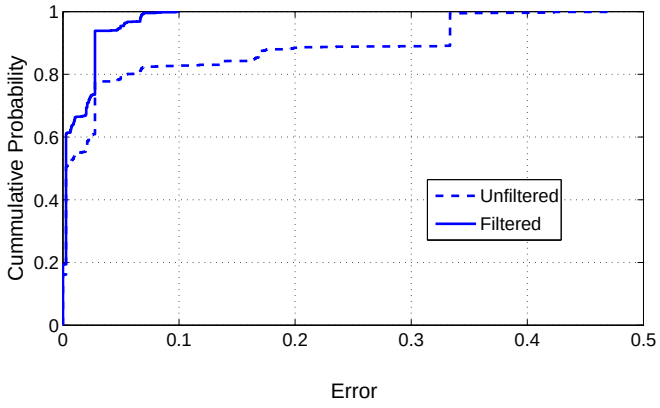


Figure 5.9: The plots for error in estimating the query costs using QCE. The dashed line represents all queries in the workload, while the solid line represents queries with higher than 5 unit cost. The unit of the cost is same as the one used by SQL Server's optimizer to compare the alternative plans.

below 5 optimizer cost units. We plot the CDF for errors after removing those light queries, which reduces the workload size by 52%. The solid line in Figure 5.9 shows the cost estimation error for these filtered set of queries. The maximum error for the filtered queries is about 11%, and about 94% of the estimations have less than 5% error.

The inaccuracies in the light queries comes from the approximations discussed in Section 5.1.4. Since the contribution of the light queries to the total workload cost is insignificant compared to the contribution of the heavy queries–only about 4% for our workload–the inaccuracy in estimating their costs does not affect the configurations selected by our algorithm.

5.2 Online Tuning for Indexes

Before delving into the details of the online algorithm for indexes, we first demonstrate that the efficient and easy-to-implement algorithm of the previous section cannot be directly applied to the online algorithm for determining indexes. The reason is the unbounded nature of ρ. We illustrate this by using an example scenario in which I_1, and I_2 are two possible indexes in the space $\mathcal{S}$.

Let $S_1 = \{I_1, I_2\}$, $S_2 = \{I_1\}$. If B_{I_2} is the cost of building the index I_1, then $d(S_1, S_2) = B_{I_1}$, and $d(S_2, S_1) = 0$, implying that the ratio $\rho = \infty$. Therefore, an adversary can make the algorithm shift between these two states and make the online algorithm perform infinitely worse than the optimal algorithm.

Furthermore, vertical partitions require only a minimally larger footprint on the disk, when compared to the original tables. Indexes, however, need a substantially larger footprint, as they duplicate the data from the original table. Therefore, one needs to take into account the extra storage overhead of the indexes.

We therefore build an online algorithm OnlineITwhich is based on the famous Work Function Algorithm [9]. The algorithm incurs higher overhead but provides a tighter competitive ratio than the OnlinePD algorithm. We also introduce a mechanism to take into account the cost of storing the index on disk. We first discuss the work function algorithm, and then discuss the details of our adaption of online physical design to the WFA, and then demonstrate the effectiveness of the algorithm for the physical design problem.

5.2.1 Work Function Algorithm

Work functions are important part of the online algorithms, and play vital role in finding the competitiveness of the algorithms. The intuition behind the algorithm is that, the online algorithm needs to keep track of the optimal offline cost so far. This realization leads to the concept of "work functions", that try to follow the optimal offline algorithm with a small "regularization" term.

Remember that S is the set of all possible configurations (combination of indexes), and d defines the distance of one configuration from the other. Let s_0 be the initial state of the database, q_i be the i^{th} query in the sequence, and the cost of the query at state $s \in S$ be $cost(q_i, s)$. The entire sequence of the queries can be represented as $\sigma = q_1, q_2, \ldots, q_n$. Then the function w_i defines the

work function as follows:

$$w_{i+1}(s) = \min_{x \in S}\{w_i(s) + r_{i+1}(x) + d(x,s)\} \tag{5.4}$$

$$w_0(s) = d(s_0, s) \tag{5.5}$$

Clearly, the optimal algorithm for σ is $\min_{x \in S} w_n(x)$.

Using this work function, we now define the work function algorithm **WFA**: Suppose that the algorithm is in state s_i after processing q_i, then on receiving the query q_{i+1} the algorithm moves to a state $s_{i+1} = arg\min_x\{w_{i+1}(x) + d(s_i,x)\}$, and $w_{i+1}(s_{i+1}) = w_i(s_{i+1}) + r_{i+1}$.

It has been proven that the above algorithm is only $2|S| - 1$ competitive [62].

5.2.2 OnlineIT

We now compute the cost of answering the query $query\_cost(q_i,x)$. The cost of the answering the query includes the cost of actually running the query, and the cost of the keeping the structures in s_{i+1} till the arrival of q_{i+1}. Therefore the cost is

$$r_{i+1}(x) = query\_cost(q_{i+1},x) + maint\_cost(s_i)$$

Here the *query_cost* is the cost of the query to execute in the configuration x, and the *maint_cost* is the cost of maintaining the structure in the database. Typically the *query_cost* is the cost of running the query in the database and measured by invoking the optimizer and retrieving the plan, or invoking tools like INUM to reduce the cost of optimizer invocation. The maintenance cost involves computing the rate of updates to the index structures, and the cost of storing the index on disk. Traditionally, these metrics had been separate ones, and there was no way to directly compare them. In the advent of cloud computing, however, allow us to compare them using the cost metrics

imposed by the cloud providers. Section 5.2.4 discusses the details of measuring these costs and comparing them on dollar terms.

5.2.3 Implementation

We use the architecture similar to CoPhy, we repeat it here again. We take two approach to determine the work-function: Dynamic Programming and Combinatorial Optimization Programming.

Dynamic Programming

Using the dynamic programming approach, all $s \in S$ are enumerated, and for each query the work-function for that state is computed. As one can see, the complexity of the algorithm is $O(|S| \times |I|)$ for each step. Typically, $|S|$ is as high as a million for typical analytical queries and $|I|$ is in thousands. To reduce the overhead, we first consider only $s \in S(query\_cost(q_i, s) > query\_cost(q_i, \phi)$. Using INUM the costs can be determined fast, and furthermore it can be sampled to reduce the number of configurations the algorithm needs to look at.

Combinatorial Optimization Programming

The Combinatorial Optimization Program (COP) approach removes the requirement for sampling the states, but at the cost of adding higher overhead to the computation. Recall from Chapter 4 that the *query_cost* function can be represented as a linear function with variables for the indexes. The number of such variables is linear w.r.t. to the number of indexes, therefore, does not grow combinatorially as is the case for the dynamic programming approach.

Still, for millions of queries, the overhead of solving the problem can increase beyond manageable level. We address this issue by taking the following three approaches:

1. Grouping the queries into batches. Instead of solving the work function problem for each

incoming query, we group a set of b queries into one batch to do the index tuning. This allows us to reduce the impact of the COP solver overhead for each query. Note that, the batching does not affect the competitiveness of the algorithm after the batches are complete. It merely differs the competitiveness to the end of the batch.

2. A window is used to decide on the physical design by looking at only last w batches, rather than the complete query history. If $w = 1$, then this resembles the greedy algorithm which tries to optimize for only one query. For $w = \infty$, this is the full work function algorithm. We experimentally show that by keeping the w value at reasonable level we get structures which are close to the full work function algorithm, but are much faster in execution.

3. We update the COP for each new query, rather than recreate the COP for each new set of queries. As discussed earlier in Chapter 4, the COP solvers can reuse the earlier computations to speed up the incremental updates to the problem. Using this advantage, we only remove a batch at the beginning of the window and remove add the current batch to the problem before solving it.

Using above three techniques we show that using the COP solver is more efficient than using the dynamic programming for a space of the parameter space.

5.2.4 Cloud Cost Model

In this section we describe the infrastructure details of the cloud setting and we analyze the cost for building cache structures and executing query plans.

Cloud Infrastructure

Our system is targeted towards an architecture shown in Figure 5.10. The user requests for query execution from Internet and contacts the Coordinator node. The Coordinator node distributes the

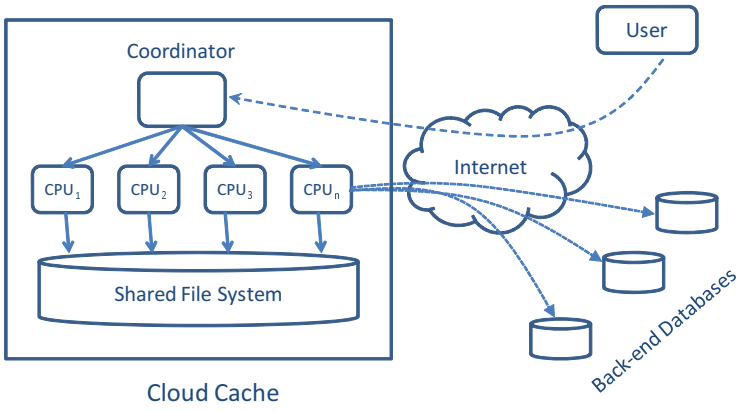


Figure 5.10: The Cloud DBMS Architecture

query to the appropriate CPU node, or to the back-end databases. We assume that the cloud infrastructure provides unlimited amount of storage space, CPU nodes, and very high speed intra-cloud networking. We also assume that the CPU nodes in the architecture are all identical to each other. Compared to TCP bandwidth on the Internet, the inter-system bandwidth is orders of magnitude faster and we ignore the overhead associated with it. Our storage system is based on a clustered file system, such as, [29], where the disk blocks are replicated and stored close the CPU nodes accessing them. With this infrastructure we can assume that the virtual disk is a shared resource for all the CPU nodes in the cache.

Cost of Structures

To speed up the queries we also implement indexes for the queries. In the future we plan to expand the infrastructure in order to gain extra performance from cached materialized views, similar to [73], or partial columns, similar to [70]. Moreover, additional CPUs are employed to speed up queries. Therefore, the cache needs to decide on implementing three different types structures to speed up the queries and maintain them, i.e., 1) CPU nodes N, 2) table columns T, and 3) indexes I. As discussed in Section 4.3 the cache needs to spend money for building these structures and maintain them. For all three structures we define the building structure $Build_S$, as well as the

maintenance structure $Maint_S$, for $S \in \{N, T, I\}$.

For CPU nodes, we exploit the scalability of cloud infrastructure and dynamically boot up a system on demand. If b is the time it takes to boot a system and u is the cost of using a system per unit time, then the cost of building the CPU node, N structure is constant:

$$Build_N(N) = b \cdot u \tag{5.6}$$

and the cost of maintaining the node per unit time is also constant:

$$Maint_N(N) = c \tag{5.7}$$

For a table column, T, the building cost $Build_C(T)$ is the cost of transferring T from the back-end database and combining it with the current columns in the cache. For simplicity, we ignore the cost of integrating T into the existing table in the cache. Therefore the cost of building a table column is primarily the cost of transferring the respective data over the network. Therefore the cloud has to pay for the bandwidth used to transfer that data, and the CPU time taken to manage that transfer. If f_n is the fraction of CPU taken to manage transfer, and t, l are the throughput and latency of the network, the total cost for building a column C in the cache is:

$$Build_T(T) = f_n \cdot (l + \frac{size(T)}{t}) + size(T) \cdot c_b \tag{5.8}$$

where c_b is the cost of transferring a byte across the network. The maintenance cost of the table column is just the cost of using disk space in the cloud, so if c_d is the cost of disk storage in the cloud, then the maintenance cost for T is:

$$Maint_T(T) = size(T) \cdot c_d \tag{5.9}$$

For indexes, the building cost involves fetching the data across the network and then building the index on the cache. Since sorting is the most important step in building an index, we approximate the cost of building an index to the cost of sorting the indexed columns. For example, the cost for

building an index $I(A,B,C)$ on table T using columns A,B, and C is approximated to the cost of running the following query:

```
Q = select A, B, C from T order by A, B, C
```

The total cost building an index is therefore,

$$Build_I(I) = C_e(P_Q) + \sum_{C \in I \wedge C \notin Cache} Build_I(I) \tag{5.10}$$

where P_Q is the query plan for the above query. The function $C_e(P_Q)$, is the function that determines the cost of running a query Q in the cache based on a specific plan P_Q, and is described in Section 5.2.4. We assume that the cloud databases in the current setting are static, therefore there is no need for index updating. Hence, the maintenance cost for an index is:

$$Maint_I(I) = size(I) \cdot c_d \tag{5.11}$$

Cost for Running Queries

Since the cloud DBMS should consider many possible plans, i.e. design configurations, for running a query, accurate and fast estimation of the cost associated with each plan is very important. The execution time of a query C_e is estimated based on a respective query plan P_Q. For the current cloud setting, we assume that the query runs completely either in the back-end or in the cache. In other words, we do not consider plans that run partially in the back-end and then transfer the partially computed data to the cache in order to complete the execution.

Concerning queries that run completely in the cache, estimation of the execution cost is determined based a plan that is suitable for the cache. If the total cost of running the query is q_{tot}, and total I/O cost for in the execution plan is io_{tot}, the cost of running the query is:

$$C_{e_C}(P_Q) = l_{cpu} \cdot f_{cpu} \cdot q_{tot} \cdot c + f_{io} \cdot io \cdot io_{tot} \tag{5.12}$$

where l_{cpu} is a factor that indicates the overload of the CPU node, and, f_{cpu} and f_{io} are factors that convert the numbers reported by the execution plan to actual CPU time and actual IO operations,

respectively. If these factors are stable their values can be estimated by running a fixed set of simple queries and plotting the actual CPU time and logical disk reads. If these factors change, their values are determined by the actual execution plan after running the queries??.

For network queries, we estimate the cost as the total cost of running in the back-end database and transferring the result to the cache.

$$C_{e_N}(Q) = C_{e_C}(Q) + f_n \cdot (L + \frac{S(Q)}{t}) + S(Q) \cdot c_b \tag{5.13}$$

The function $S(Q)$ determines the size of the results for the query Q, and the values t, f_n, c_b, and l are defined in Section 5.2.4.

5.2.5 Experimental Setup

We use the experimental setup of CoPhy to run the experiments for the above mentioned algorithms.

Workload We use a work generated from 7 TPC-H query templates and containing 10000 queries. The queries are arranged such that they mimic the evolution demonstrated by the original SDSS workload (as shown in Figure 5.4). We use this workload, since current INUM implementation is not mature enough to process complex query structures, and deep hierarchy of function used in the SDSS workload.

Batch Size (b): The number of queries used for each invocation of the COP Solver. We experiment with batch sizes of 1, 10, 50, 100 and determine that at the batch size of 50, the overhead of invoking the COP solver is dominated by the actual cost of solving the optimization problem. Therefore, we use $b = 50$ as the default value in all our experiments.

Window Size (w): The number of queries kept for computing the work function algorithm. We experiment with window sizes of 100, 500, and 1000 and 10000 (i.e., effectively disabling the windowing).

Query Arrival Interval (r): We study the effect of the index maintenance cost by changing the query arrival interval. For larger interval, the maintenance cost increases, and for smaller interval the maintenance cost is low. We experiment with three different value of r – 10s, 30s, and 60s.

We use a dual-core Xeon machine with a commercial DBMS (*System2*) on a TPCH database of size 1GB.

5.2.6 Experimental Results

We first demonstrate that using a window on the work function allows us to reduce the overhead of its computation without sacrificing the workload performance. Then we show the effect of maintenance cost on the online physical design problem.

Effect of Window Size

In this experiment, we compare the performance of the algorithms on various window sizes for the COP approach. It also compares the performance of the COP approach with the dynamic programming approach and the offline approach. In offline approach we optimize the entire workload only once, and use the suggested indexes for all queries in the workload. For this experiment we ignore the maintenance cost of the indexes.

Figure 5.11 shows the average query cost for the all the batches in the workload. The figure shows that using an online approach is beneficial for an evolving workload, as the performance of the offline tuning algorithm is worse than the online algorithm for most of the batches, except for the first batch, where the online algorithm does not begin to optimize before the batch of the query has arrived.

Figure 5.12 compares the average overhead of the tuning algorithms for each query along with the actual execution time of the queries using the suggested indexes. In our workload, the dynamic

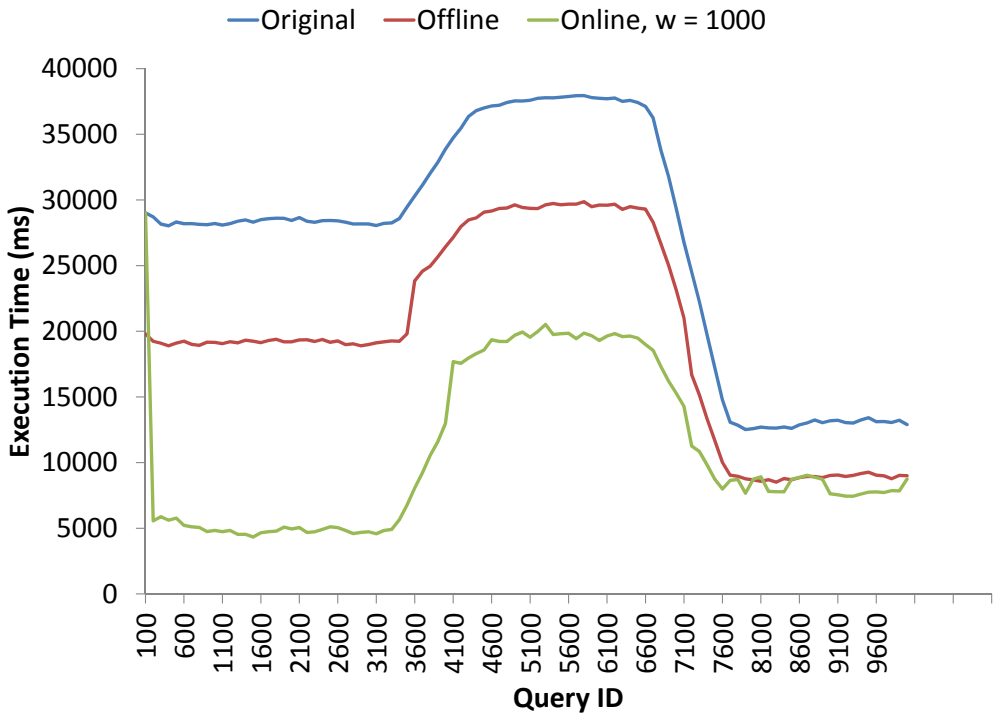
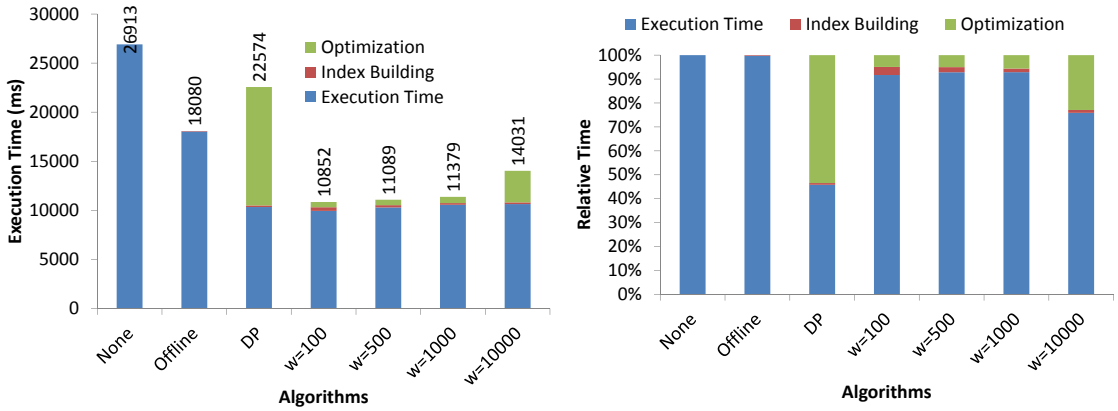


Figure 5.11: The response times for the original workload, the offline algorithm, and COP algorithm with $w = 100$.

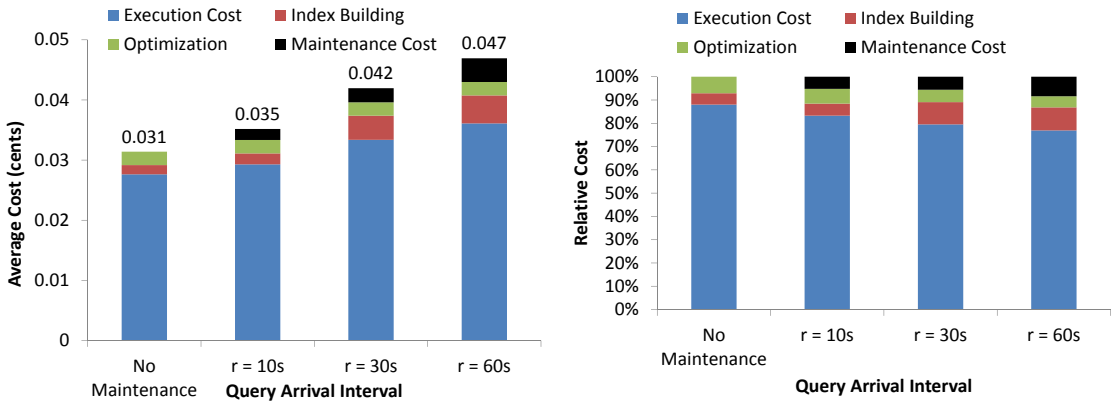
program has to check approximately 6.210^6 combinations of states and indexes for each batch. Therefore, it incurs a very high optimization overhead, even more than using the COP without windowing. The overhead of COP approach is about 5% for seconds for most window sizes, which is reasonable considering that reduce the execution time of the workload by a factor of 2.5. The overhead the COP approach goes up to 23% for $w = 10000$, since the COP requires about 6 minutes to solve for each batch. Thus, using the COP for a reasonable window size of 100 provides ideal balance between the performance and overhead.



(a) Comparison of the absolute values of the time spent.

(b) Comparison of the relative values of the time spent.

Figure 5.12: The response times for the workload for various online optimization algorithms.



(a) Comparison of average dollar cost for different components.

(b) Comparison of relative cost for different components.

Figure 5.13: The average cost of running the queries on the system.

Effect of Maintenance Cost

To study the effect of maintenance cost, we use the cost model in from Aamazon's EC2 cloud service and convert all the costs into dollars as discussed in Section 5.2.4. We then vary the arrival rate of the queries. Since the database on which we run the experiments is relatively small, we increase the maintenance cost by three orders of magnitude to make it comparable to a cloud-scale maintenance cost. In this experiment, we set w to be 100.

Figure 5.13 compares the costs of different components of the algorithms for various query arrival intervals. As the query arrival interval increases the cost of building the indexes and maintaining them increases. Since the maintenance cost of the indexes forces the algorithm to remove indexes, the build cost also increases with increasing arrival rates. But, they do not increase proportionally to the arrival interval, as the algorithm optimizes for the total cost in this scenario.

5.3 Related Work

As we discuss in earlier chapters, automated physical design tools can improve cache performance by using query workload to decide which attributes to group together and output the best the hybrid model. Early work on automated physical design [3, 25, 34, 49, 49, 52] derived affinity measurements from a given workload as a measure for grouping columns together. Columns are grouped by applying clustering algorithms on the affinity values. However, affinity values are decoupled from actual I/O cost, and thus are poor predictors of the best hybrid model. Recently, cost estimates from the optimizer or analytical cost-based models that capture the I/O of database operations are used to evaluate attribute groupings [3, 22, 52]. For instance AutoPart[52] interfaces with a commercial optimizer to obtain cost estimates for queries. Existing physical design solutions are offline, i.e., they assume a priori knowledge of the entire workload stream and therefore provide a single, static physical design for the entire workload.

Current research [13, 63] emphasizes the need for automated design tools that are *always-on* and as queries arrive continuously adapt the physical design to changes in the workload [4]. Quiet [59] describes an incremental algorithm for index selection which is not fully integrated with the optimizer. In COLT [63], the authors present an alternative heuristic algorithm which relies heavily on the optimizer. In addition, they do not take into account transition costs. Bruno et. al. present a formal approach to online index selection [13]. A 3-competitive algorithm is proposed which adapts to workload changes as well as accounts for transition costs. However, the algorithm applies to only two physical design alternatives: creation of an index and dropping an index on a table. For multiple indexes, similar to COLT, they present a different heuristic incremental algorithm.

In this chapter, we describe a general framework that evaluates multiple physical design alternatives. We are concerned with sudden changes in the workload and therefore use online algorithms that provide guarantees on being able to adapt with changes in workload. To the best of our knowledge this is the first framework that evaluated multiple physical design alternatives in a

formal way.

5.4 Conclusion

In this chapter, we have presented a workload adaptive physical design tool that can help DBAs who manage large database installations. This tool takes the guess work out of a DBA's job by running continuously and adapting to workload and database changes.

Our tool is general in that it does not depend upon the distribution properties of the incoming workload but assumes the worst possible workload. Therefore it adapts to drastic changes in the workload. This adaptiveness comes from the competitive online algorithm. While an adaptive algorithm is necessary for the tool to be general it has to be efficient also. We have supported our tool with efficient cost estimation modules. This makes our tool practical for real world database installations such as proxy caches.

Our experimental evaluation shows that the tool not only outperforms existing offline methods, it adapts quite close to the special methods designed specifically for a given workload. We show that our tools do not hurt DBMS performance but maintains it.

Chapter 6

Conclusion

In this thesis, we described a novel approach to physical design tuning. First, we propose a cost model that can be modeled mathematically. The key insight is, instead of fully modeling the query optimizer or treating it as a black-box, we partially model to take advantage of both the accuracy of a black-box model and the flexibility of the full model. We reuse the query optimizer's computations to model the dependence of the query cost on its data access costs. This approach allows the cost model to be as accurate as the invoking the optimizer directly and build a straightforward mathematical model to be exploited in search algorithms.

Second, we exploited this cost model to formulate the physical design problem as a compact combinatorial optimization problem. The compactness of the formulation results in fast identification of near optimal design features, while scaling to thousands of queries and candidate indexes. Furthermore, the formulation enables many novel features in the physical design tool, such as interactivity, predictability, and generality.

In this thesis, we also developed physical design algorithms for online workloads that provide competitiveness guarantees. We first developed a very low overhead algorithm using graph traversal technique, that works on highly resource constrained environments. We then developed a

general algorithm by extending our combinatorial optimization formulation.

The application of our approach to the diverse workload types, design features, and multitudes of real-world constraints demonstrates that it is generic enough to be applied to other physical design scenarios as well. The approach of partially modeling the optimizer can be used to build useful cost models in presence of selectivity variations, or scenarios in which the underlying components are too complex to model completely. The combinatorial optimization formulation can also be used for any scenarios in which the behavior of the query optimizer needs to optimized. For example, it can be used to efficiently identify query and index interactions. Finally, the approach of using principled optimization methods by converting the database optimization problem into a mathematical optimization problem can be used for many other optimization problems inherent in the database management systems.

Bibliography

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the VLDB Conference*, 2000. 3.1.1, 2, 3.5, 4.9.1

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*, 2004. 1.1.3, 4.2

[3] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In *SIGMOD*, 2004. 3.5, 5.3

[4] Sanjay Agrawal, Eric Chu, and Vivek Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 683–694, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: http://doi.acm.org/10.1145/1142473.1142549. 1.1.3, 5.3

[5] Jair M. Babad. A record and file partitioning model. *Commun. ACM*, 20(1):22–31, 1977. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359367.359418. 1.1.1

[6] Bananno, R., Maio, D., and P. Tiberio. An Approximation Algorithm for Secondary Index Selection in Relational Database Physical Design. *The Computer Journal*, 28(4):398–405, 1985. doi: 10.1093/comjnl/28.4.398. URL `http://comjnl.oxfordjournals.org/cgi/content/abstract/28/4/398`. 1.1.1

[7] Flavio Bonfatti, Dario Maio, and Paolo Tiberio. A separability-based method for secondary index selection in physical database design. In *Methodology and Tools for Data Base Design*, pages 149–160. 1983. 1.1.1

[8] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-56392-5. 5.1.1

[9] Allan Borodin, Nathan Linial, and Michael E. Saks. An Optimal Online Algorithm for Metrical Task System. *J. ACM*, 39(4):745–763, 1992. ISSN 0004-5411. 5.1.1, 11, 5.2

[10] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, March 2004. ISBN 0521833787. URL `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0521833787`. 2, 4.5.4

[11] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *SIGMOD '05*. 3.1.1, 4, 3.2.1, 3.2.2, 3.2.3, 3.2.3, 3.5

[12] Nicolas Bruno and Surajit Chaudhuri. To tune or not to tune?: a lightweight physical design alerter. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 499–510. VLDB Endowment, 2006. ISBN 1-59593-385-9. 3.2.2, 3.2.3, 3.2.3

[13] Nicolas Bruno and Surajit Chaudhuri. An online approach to physical design tuning. In *IEEE 23rd International Conference on Data Engineering (ICDE 2007)*, 2007. 1.1.3, 5.3

[14] Nicolas Bruno and Surajit Chaudhuri. Constrained physical design tuning. *PVLDB*, 1(1): 4–15, 2008. 1.1.3, 4.5

[15] A. Caprara and J. Salazar. A branch-and-cut algorithm for a generalization of the uncapacitated facility location problem. *TOP*, 1996. 4.2

[16] Alfonso F. Cardenas. Evaluation and selection of file organization—a model and system. *Commun. ACM*, 16(9):540–548, 1973. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/362342.362352. 1.1.1

[17] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 3–14. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. 1.1.3

[18] Surajit Chaudhuri and Vivek R. Narasayya. Index merging. In *Proceedings of ICDE 1999*. 3.5

[19] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL server. In *Proceedings of the VLDB Conference*, 1997. 1.1.3, 3.1.1, 2, 3.2.1, 3.2.5, 3.2.7, 3.2.8, 3.2.8, 3.5, 4.9.1

[20] Surajit Chaudhuri, Mayur Datar, and Vivek Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1313–1323, 2004. ISSN 1041-4347. doi: http://doi.ieeecomputersociety.org/10.1109/TKDE.2004.75. 3.1, 3.5, 4.1

[21] Sunil Choenni, Henk M. Blanken, and Thiel Chang. On the automation of physical database design. In *SAC '93: Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing*, pages 358–367, New York, NY, USA, 1993. ACM. ISBN 0-89791-567-4. doi: http://doi.acm.org/10.1145/162754.162932. 1.1.2

[22] W. W. Chu and I. T. Ieong. A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems. *IEEE Trans. Software Eng.*, 19(8):804–812, 1993. ISSN 0098-5589. 5.3

[23] Douglas Comer. The difficulty of optimum index selection. *ACM Trans. Database Syst.*, 3 (4):440–445, 1978. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/320289.320296. 1.1.1

[24] Mariano P. Consens, Denilson Barbosa, Adrian Teisanu, and Laurent Mignet. Goals and benchmarks for autonomic configuration recommenders. In *SIGMOD '05*. 3.2.7

[25] Douglas W. Cornell and Philip S. Yu. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE Trans. Software Eng.*, 16(2):248–258, 1990. ISSN 0098-5589. 5.3

[26] Christopher E. Dabrowski, David K. Jefferson, John V. Carlis, and Salvatore T. March. Integrating a knowledge-based component into a physical database design system. *Information & Management*, 17(2):71 – 86, 1989. ISSN 0378-7206. doi: DOI:10.1016/0378-7206(89)90009-8. URL `http://www.sciencedirect.com/science/article/B6VD0-45M2R9B-3B/2/da04ff4b3c3e549f1bdd80da2aa30074`. 1.1.2

[27] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. Automatic sql tuning in oracle 10g. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1098–1109. VLDB Endowment, 2004. ISBN 0-12-088469-0. 1.1.3

[28] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13:91–128, 1988. 1.1.2

[29] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003. 5.2.4

[30] A. Ghosh, J. Parikh, V. Sengar, and J. Haritsa. Plan selection based on query clustering. In *VLDB*, 2002. 3.5

[31] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. 2.4

[32] Aberdeen Group. Dbms cost breakdown, 1998. URL `http://relay.bvk.co.yu/progress/aberdeen/aberdeen.htm`. 1

[33] G.Valentin, M.Zuliani, D.Zilio, and G.Lohman. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of ICDE'00*. 3.1.1, 3.2.8, 3.5, 4.2

[34] Michael Hammer and Bahram Niamir. A Heuristic Approach to Attribute Partitioning. In *SIGMOD*, 1979. 5.1.5, 5.1.7, 5.3

[35] C. Heeren, H. V. Jagadish, and L. Pitt. Optimal indexes using near-minimal space. In *PODS*, 2003. 4.2, 4.8.1

[36] Jeffrey Alan Hoffer. *A clustering approach to the generation of subfiles for the design of a computer data base.* PhD thesis, Ithaca, NY, USA, 1975. 1.1.1

[37] A. Hulgeri and S. Sudarshan. Parametric query optimization for linear and piecewise linear cost functions. In *VLDB '02*. 3.5

[38] Arvind Hulgeri and S. Sudarshan. AniPQO: Almost non-intrusive parametric query optimization for nonlinear cost functions. In *Proceedings of the VLDB Conference*, 2003. 3.2.6, 3.5

[39] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 413–424, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: http://doi.acm.org/10.1145/1247480.1247527. 1.1.3

[40] M. Y. L. Ip, L. V. Saxton, and V. V. Raghavan. On the selection of an optimal set of indexes. *IEEE Trans. Softw. Eng.*, 9(2):135–143, 1983. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1983.236458. 1.1.1

[41] Hideaki Kimura, George Huo, Alexander Rasin, Samuel Madden, and Stanley B. Zdonik. Correlation maps: a compressed access method for exploiting soft functional dependencies. *Proc. VLDB Endow.*, 2(1):1222–1233, 2009. ISSN 2150-8097. 1.1.3

[42] W. F. King. On the selection of indices for a file. *IBM Research Journal*, 1341, 1974. 1.1.1

[43] Maxim Kormilitsin, Rada Chirkova, Yahya Fathi, and Matthias Stallmann. View and index selection for query-performance improvement: quality-centered algorithms and heuristics. In *CIKM*, pages 1329–1330, 2008. 4.2, 4.8.1

[44] Jozef Kratica, Ivana Ljubic, and Dušan Tošic. A genetic algorithm for the index selection problem. In *EvoWorkshops'03: Proceedings of the 2003 international conference on Applications of evolutionary computing*, pages 280–290, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00976-0. 1.1.3

[45] Sam S. Lightstone, Toby J. Teorey, and Tom Nadeau. *Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007. ISBN 0123693896. 1

[46] Vincent Y. Lum and Huei Ling. An optimization problem on the selection of secondary keys. In *ACM '71: Proceedings of the 1971 26th annual conference*, pages 349–356, New York, NY, USA, 1971. ACM. doi: http://doi.acm.org/10.1145/800184.810505. 1.1.1

[47] Tanu Malik, Xiaodan Wang, Randal Burns, Debabrata Dash, and Anastasia Ailamaki. Automated Physical Design in Database Caches. In *SMDB*, 2008. 1, 5.1.7

[48] Tanu Malik, Xiaodan Wang, Debabrata Dash, Amitabh Chaudhary, Anastasia Ailamaki, and Randal Burns. Online physical design in proxy caches. Technical Report `http://hssl.cs.jhu.edu/fedcache/onlinepd`, John Hopkins University, 2008. 5.1.7

[49] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical Partitioning Algorithms for Database Design. *ACM Trans. Database Syst.*, 9(4):680–710, 1984. ISSN 0362-5915. 5.1.7, 5.3

[50] F. Palermo. A quantitive approach to the selection of secondary indexes. *IBM*, 1970. 1.1.1

[51] Stratos Papadomanolakis. *Large-scale data management for the sciences*. PhD thesis, CMU, 2007. 4.2, 4.3.2, 4.8.1, 4.9.3

[52] Stratos Papadomanolakis and Anastassia Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM*, 2004. 5.1.5, 5.3

[53] Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. Efficient use of the query optimizer for automated physical design. In *VLDB '07*, pages 1093–1104. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. 4.8.1, 13, 4.9.3

[54] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 558–569. ACM Press, 2002. ISBN 1-58113-497-5. doi: http://doi.acm.org/10.1145/564691.564757. 3.5

[55] Naveen Reddy and Jayant R. Haritsa. Analyzing plan diagrams of database query optimizers. In *Proceedings of VLDB*, 2005. 3.5

[56] Steve Rozen and Dennis Shasha. A framework for automating physical database design. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 401–411, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. ISBN 1-55860-150-3. 1.1.2

[57] A Runapongsa, Jignesh M. Patel, and Rajesh Bordawekar. Xist: an xml index selection tool. In *In Proc. of 2nd Intl. XML Database Symp. (XSym*, 2004. 1.1.3

[58] Agrawal S., Chaudhuri S., Kollar L., and Narasayya V. Index tuning wizard for microsoft sql server 2000. Technical report, Microsoft Research, 2000. 1.1.3

[59] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. Quiet: continuous query-driven index tuning. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 1129–1132. VLDB Endowment, 2003. ISBN 0-12-722442-4. 1.1.3, 5.3

[60] Mario Schkolnick. Secondary index optimization. In *SIGMOD '75: Proceedings of the 1975 ACM SIGMOD international conference on Management of data*, pages 186–192, New York, NY, USA, 1975. ACM. doi: http://doi.acm.org/10.1145/500080.500106. 1.1.1

[61] Mario Schkolnick. A survey of physical database design methodology and techniques. In *VLDB'1978: Proceedings of the fourth international conference on Very Large Data Bases*, pages 474–487. VLDB Endowment, 1978. 1.1.1

[62] Karl Schnaitter and Neoklis Polyzotis. Semi-automatic index tuning: Keeping dbas in the loop. *CoRR*, abs/1004.1249, 2010. 5.2.1

[63] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. Colt: continuous on-line tuning. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 793–795, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. doi: http://doi.acm.org/10.1145/1142473.1142592. 1.1.3, 5.3

[64] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. Index interactions in physical design tuning: modeling, analysis, and applications. *Proc. VLDB Endow.*, 2(1):1234–1245, 2009. ISSN 2150-8097. 3.4.1, 2

[65] sdss. The Sloan Digital Sky Survey. URL `http://www.sdss.org`. 5, 5.1.5, 5.1.7

[66] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1979. 3.2.1, 3.2.6, 5.1.4

[67] David Simmen, Eugene Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 57–67, New York, NY, USA, 1996. ACM. ISBN 0-89791-794-4. doi: http://doi.acm.org/10.1145/233269.233320. URL `http://doi.acm.org/10.1145/233269.233320`. 3.3.3

[68] Alan Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, page 101, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0506-6. 1.1.3

[69] Michael Stonebraker. The choice of partial inversions and combined indices. *International Journal of Parallel Programming*, 3(2):167–188, 1974. 1.1.1

[70] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: a wide-area distributed database system. *The VLDB Journal*, 5(1):048–063, 1996. ISSN 1066-8888. doi: http://dx.doi.org/10.1007/s007780050015. 5.2.4

[71] Zohreh Asgharzadeh Talebi, Rada Chirkova, Yahya Fathi, and Matthias Stallmann. Exact and inexact methods for selecting views and indexes for olap performance improvement. In *EDBT*, pages 311–322, 2008. 4.2

[72] `http://dbtools.cs.cornell.edu/norm_index.html`. The database normalization tool. 1

[73] X. Wang, T. Malik, R. Burns, S. Papadomanolakis, , and A. Ailamaki. A Workload-Driven Unit of Cache Replacement for Mid-Tier Database Caching. In *DASFAA*, 2007. 5.1.5, 5.2.4

[74] Kyu young Whang, Gio Widerhold, and Daniel Sagalowicz. Separability an approach to physical database design. In *IEEE Transcation on Computers*. 1.1.1

[75] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *VLDB*, 2004. 1.1.3, 3.1.1, 3.5