# Automated Database Design for Large-Scale Scientific Applications

Stratos Papadomanolakis

May 2007

Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

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

**Thesis Committee:**
Anastassia Ailamaki, Chair
Christos Faloutsos
David R. O'Hallaron
Gerd Heber, Cornell Theory Center, Cornell University

Copyright © 2007 Stratos Papadomanolakis

This work was supported in part by NSF through grants IIS-0431008, IIS-0429334, IIS-0328740, CCF-0326453, IIS-0133686, and CCF-0205544, as well as through the NASA AISR Program. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

# Contents

**1 Overview** **9**

1.1 Challenges in Scientific Data Management . . . . . . . . . . . . . . . 12

1.1.1 Relational Database Support for Scientific Applications . . . . 13

1.1.2 Support for New Data Organizations . . . . . . . . . . . . . . 17

1.2 Designing for Performance . . . . . . . . . . . . . . . . . . . . . . . . 20

1.2.1 Addressing Limitations of Existing Systems . . . . . . . . . . 20

1.2.2 Efficient Use of the Query Optimizer for Automated Database Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

1.2.3 An Integer Linear Programming Approach to Automated Database Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

1.2.4 Workload Driven Schema Partitioning . . . . . . . . . . . . . 26

1.3 Efficient Query Processing for Simulations . . . . . . . . . . . . . . . 28

**2 Related Work** **31**

2.1 Systems for Large-Scale Scientific Data Management . . . . . . . . . 31

2.2 Managing Scientific Data using a DBMS . . . . . . . . . . . . . . . . 32

2.3 Non-DBMS Architectures . . . . . . . . . . . . . . . . . . . . . . . . 35

2.4 Automated Database Design . . . . . . . . . . . . . . . . . . . . . . . 36

2.4.1 Index Selection . . . . . . . . . . . . . . . . . . . . . . . . . . 36

2.4.2 Materialized Views and Partitioning Selection . . . . . . . . . 41

2.4.3 Other Research on Physical Design . . . . . . . . . . . . . . . 43

2.4.4 Theoretical Approaches . . . . . . . . . . . . . . . . . . . . . . 45

2.5 Multidimensional Indexing . . . . . . . . . . . . . . . . . . . . . . . . 46

**3 Efficient Use of the Query Optimizer in Automated Database Design 49**

3.1 Introduction . . . . . . . . . . 49

3.1.1 Contributions . . . . . . . . . . 51

3.2 INUM Fundamentals . . . . . . . . . . 53

3.2.1 Setup: The Index Selection Session . . . . . . . . . . 54

3.2.2 Reasoning About Optimizer Output . . . . . . . . . . 55

3.3 INUM Overview . . . . . . . . . . 60

3.4 Using Cached MHJ Plans . . . . . . . . . . 63

3.4.1 A Formula for Query Cost . . . . . . . . . . 63

3.4.2 Mapping Configurations to Optimal Plans . . . . . . . . . . 65

3.5 Computing the INUM Space . . . . . . . . . . 69

3.6 Extending the INUM . . . . . . . . . . 72

3.6.1 Modeling NLJ Plans . . . . . . . . . . 72

3.6.2 Extending INUM with NLJ Plans . . . . . . . . . . 74

3.7 Experimental Setup . . . . . . . . . . 77

3.8 Experimental Results . . . . . . . . . . 78

3.8.1 TPCH15 Results . . . . . . . . . . 79

3.8.2 NREF Results . . . . . . . . . . 82

3.9 Conclusion . . . . . . . . . . 83

**4 An Integer Linear Programming Approach to Automated Database Design 85**

4.1 Introduction . . . . . . . . . . 85

4.2 An ILP Model for Index Selection . . . . . . . . . . 89

4.2.1 Mathematical Formulation . . . . . . . . . . 90

4.2.2 Supporting Updates & Clustered Indexes . . . . . . . . . . 93

4.3 An ILP-based Index Selection Tool . . . . . . . . . . 94

4.3.1 ILP Solver . . . . . . . . . . 96

4.3.2 The Index Usage Model . . . . . . . . . . 97

4.3.3 Candidate Selection . . . . . . . . . . 99

4.3.4 Combination Selection . . . . . . . . . . . . . . . . . . . . . . . . 103
4.4 Approximate Cost Estimation . . . . . . . . . . . . . . . . . . . . . . . 110
4.5 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
4.6 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
4.7 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116

**5 AutoPart: Workload-Aware Schema Design for Large Scientific Applications 117**

5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
5.2 Workload-based Data Partitioning . . . . . . . . . . . . . . . . . . . . . 120
5.3 The AutoPart Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . 123
5.3.1 Terminology . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
5.3.2 Algorithm Overview . . . . . . . . . . . . . . . . . . . . . . . . 124
5.3.3 Categorical Partitioning . . . . . . . . . . . . . . . . . . . . . . 126
5.3.4 Composite Fragment Generation . . . . . . . . . . . . . . . . . 128
5.3.5 Greedy Fragment Selection . . . . . . . . . . . . . . . . . . . . 129
5.3.6 Cost evaluation: Cost models . . . . . . . . . . . . . . . . . . . 130
5.3.7 Pairwise Merging . . . . . . . . . . . . . . . . . . . . . . . . . . 131
5.4 System Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
5.5 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
5.6 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
5.6.1 Evaluation of Partitioning . . . . . . . . . . . . . . . . . . . . . 135
5.6.2 Indexing a Partitioned Schema . . . . . . . . . . . . . . . . . . 137
5.7 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139

**6 Efficient Query Processing on Unstructured Tetrahedral Meshes 141**

6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
6.1.1 Querying Simulation Datasets . . . . . . . . . . . . . . . . . . . 141
6.1.2 Our Approach and Contributions . . . . . . . . . . . . . . . . . 144
6.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146

6.3 Traditional Indexing on
Tetrahedral Meshes . . . . . . . . . . . . . . . . . . . . . . . . . 147

6.3.1 R-Tree Based Techniques . . . . . . . . . . . . . . . . . . . . . 147

6.3.2 Clipping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 150

6.3.3 Z-Order based techniques . . . . . . . . . . . . . . . . . . . . 151

6.4 Directed Local Search . . . . . . . . . . . . . . . . . . . . . . . . . 153

6.4.1 Algorithm Overview . . . . . . . . . . . . . . . . . . . . . . . 153

6.4.2 Proximity Search . . . . . . . . . . . . . . . . . . . . . . . . . 155

6.4.3 Representing Element Adjacency . . . . . . . . . . . . . . . . 158

6.4.4 DLS Generalization . . . . . . . . . . . . . . . . . . . . . . . . 161

6.5 Graph-based clustering for
Tetrahedral Mesh Data . . . . . . . . . . . . . . . . . . . . . . . . 162

6.5.1 Graph Partitioning and I/O . . . . . . . . . . . . . . . . . . . 162

6.5.2 Feature-Based Clustering . . . . . . . . . . . . . . . . . . . . . 163

6.6 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . . . . 164

6.6.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . 164

6.6.2 Datasets and Queries . . . . . . . . . . . . . . . . . . . . . . . 166

6.6.3 Performance Metrics . . . . . . . . . . . . . . . . . . . . . . . 166

6.7 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . 167

6.7.1 Range Query Performance . . . . . . . . . . . . . . . . . . . . 167

6.7.2 Point Query Performance . . . . . . . . . . . . . . . . . . . . 170

6.7.3 Feature Clustering: Heart Model . . . . . . . . . . . . . . . . 172

6.7.4 Feature Clustering: Earthquake model . . . . . . . . . . . . . 173

6.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 174

**7 Conclusion** **177**

7.1 Support for New Data Organizations . . . . . . . . . . . . . . . . . 178

7.2 Automated Database Design for Large-Scale Scientific Databases . . . 180

## Abstract

The need for large-scale scientific data management is today more pressing than ever, as modern sciences need to store and process terabyte-scale data volumes. Traditional systems, relying on filesystems and custom data access and processing code do not scale for multi-terabyte datasets. Therefore, supporting today's data-driven sciences requires the development of new data management capabilities.

This Ph.D dissertation develops techniques that allow modern Database Management Systems (DBMS) to efficiently handle large scientific datasets. Several recent successful DBMS deployments target applications like astronomy, that manage collections of objects or observations (e.g. galaxies, spectra) and can easily store their data in a commercial relational DBMS. Query performance for such systems critically depends on the *database physical design*, the organization of database structures such as indexes and tables. This dissertation develops algorithms and tools for *automating* the physical design process. Our tools allow databases to tune themselves, providing efficient query execution in the presence of large data volumes and complex query workloads.

For more complex applications dealing with multidimensional and time-varying data, standard relational DBMS are inadequate. Efficiently supporting such applications requires the development of novel indexing and query processing techniques. This dissertation develops an indexing technique for *unstructured tetrahedral meshes*, a multidimensional data organization used in finite element analysis applications. Our technique outperforms existing multidimensional indexing techniques and has the advantage that can easily be integrated with standard DBMS, providing existing systems with the ability to handle spatial data with minor modifications.

**Acknowledgments**

First and foremost I thank my advisor Anastasia (Natassa) Ailamaki for her invaluable support during the 6 years of my Ph.D. Natassa introduced me to database and systems research in general. She spent countless hours with me, one weekly meeting after the other and kept me focused on the real and interesting problems all along. She taught me how to write and give presentations, introduced me to important members of the database community and through her example I learned how work professionally. She supported me through all kinds of difficulties. One could not wish for a better advisor.

Next, I thank my committee members. Professor Christos Faloutsos for his feedback and encouragement during difficult times. Professor David R. O'Hallaron for introducing me to scientific computing applications and for providing invaluable direction at the early stages of this dissertation. Dr. Gerd Heber for spending an enormous amount of time with me in person, on the phone or writing emails and explaining his research. His work had a pivotal role in this dissertation and helped me keep my work real and practical.

I owe a lot to my collaborators: Professor Babak Falsafi and Tom Wenisch. I feel lucky I worked with them during my first years at CMU. Tiankai Tu and Julio Lopez for providing help and guidance in the early stages of this dissertation. It was always a pleasure working with them about database problems related to their research. Professor Greg Ganger for his guidance. The members and staff of the Parallel Data Lab, for providing the best possible environment and support for systems research. The annual PDL retreats and visit days were an invaluable experience. GO PDL!!

Also Minglong Shao, Steve Schlosser and Jiri Schindler:great researchers and people, made me really look forward to our weekly FATES meetings. Randal Burns, Tanu Malik and Xiaodian Wang from Johns Hopkins University: Their work on mid-tier caching for scientific applications allowed me to obtain a broader perspective on the requirements of scientific data management. Last but not least, Debabrata Dash: Without his skill and determination, the work on automated database design would simply not exist.

My friends (who have not been listed as collaborators above): Stavros Harizopoulos, Spiros Papadimitriou, Yiannis Koutis, Vlad Shkapenyuk, Kivanc Sabirli. Ippokratis Pandis, Vaggelis Vlachos, Thodoris Strigkos, Mike Abd-El-Malek and Nikos Hardavellas. Minglong Shao for always smiling. I hope we keep in touch. I wish them best of luck.

Also, all the other interesting folks that I've met so far, that taught me new things about people's characters, that I did not really want to know (but apparently I needed to).

Finally, the people without whom none of this would have happened. My parents, my brother and Bianca Schroeder. I owe them everything. THANK YOU.

# Chapter 1

# Overview

Science is becoming increasingly data intensive. Advanced instrumentation, experimental infrastructures and simulation capabilities provide scientific disciplines with unprecedented volumes of data, that is used to obtain new insights and fuel new discoveries. The increasing data volumes pose new challenges to scientific data management.

Recently, a number of senior database researchers argued that database technology has evolved and is sufficient to support the storage and querying of terabyte or petabyte-scale scientific databases [28, 27]. This position is supported by a number of recent, large-scale scientific databases being successfully deployed using commercial relational DBMS technology [40, 63, 35]. Similarly, recent research results in high-performance computing demonstrate the effectiveness of employing database techniques for data management in large simulations [70].

This dissertation contributes to the ongoing effort to support scientific applications through databases, by focusing on Database Management System (DBMS) performance. Existing DBMS embody more that 20 years of research and commercial development and provide sufficient system-level support for managing large datasets, such as optimized index implementations and scalable, parallel execution and disk

access [28]. However, building an efficient, scalable infrastructure alone is not sufficient to guarantee efficient query execution. A system whose data is properly indexed and arranged on the disk is orders of magnitude more efficient compared to a system that is equivalent in terms of hardware and database engine, but lacks the appropriate database structures, especially for complex querying patterns such as those appearing in astronomy databases (See [40] for a quantitative analysis of the performance impact of indexing).

The *physical design* of a database, the organization of its tables, indexes and materialized views, is a critical factor for query performance in large systems. For instance, implementing a set of indexes that is tailored to a specific input workload, makes optimal use of the available hardware and query execution engine and minimizes query execution time. A poorly designed database, on the other hand, wastes resources such as storage and storage bandwidth on structures that are not effective for a given workload and that also require additional work to maintain in the presence of data updates. For large data volumes, it is important to have as good designs as possible: imposing a 200% storage overhead on our database for indexes might be reasonable for small database sizes, but is a poor design choice when dealing with multi-terabyte datasets. [1]

This dissertation introduces novel tools for automating the design of a database. For a given database and query workload, the goal of automated database design is to determine a set of database structures (e.g. indexes, materialized views) that optimizes the performance of the input workload, while satisfying resource constraints such as storage. Our goal is to extend the state of the art in automated physical design algorithms by improving their quality, essentially allowing them to provide higher performance with fewer resources. Simultaneously,

[1] Such overheads are common when using modern automated design tools [61].

we improve the running times of existing approaches, by allowing them to scale and process the large and complex query workloads that are common in scientific applications.

While commercial DBMS can effectively support the requirements of large datasets with a relatively flat structure, they lack the necessary mechanisms to store and process complex, multidimensional data structures. Effectively supporting multidimensional queries is a key, universal requirement for scientific data management [66]. As a result, extending commercial DBMS with spatial indexing capabilities for scientific data has been the focus of recent research activity [29, 20]. In addition to recent approaches, there exists a significant body of earlier work on multidimensional indexing [25]. Unfortunately, none of the existing threads of research provides an indexing solution for the complex, multidimensional structures used by scientific applications. Their main problem is that previous work relies on indexing *regular* structures, either regular space decompositions [29], or approximations of more complex geometries (such as Minimum Bounding Rectangles [31]).

In this dissertation, we introduce a novel indexing and query processing technique for *unstructured tetrahedral meshes*, a multidimensional data organization typically used in simulation applications. Besides improving performance compared to existing approaches, our technique can be efficiently integrated in existing DBMS with little modification. In addition to indexing, we introduce a novel topology-based approach for multidimensional data layout, that outperforms existing approaches based on space-filling curves. While our techniques target the large class of simulation applications, they can be extended to handle more general multidimensional data, in a manner similar to [20].

This chapter presents an overview of the scientific applications mo-

tivating this thesis and their data management requirements. It details the two main challenges addressed by this thesis and outlines the main results. The thesis maintained throughout this document is the following:

*The development of novel automated database physical design tools is critical for the performance and management of large-scale scientific databases supported by relational systems. Simulation-driven applications, for which relational database support is not straightforward, benefit from the development of new and efficient indexing and query processing techniques for multidimensional data.*

## 1.1 Challenges in Scientific Data Management

We present application examples that characterize the data management requirements of modern sciences. From a database engineering perspective, we classify scientific applications in two broad categories. Our classification is not strict, rather it aims in highlighting different priorities in the design of data management solutions.

The first category comprises applications that can be mapped to the relational model in an efficient and straightforward fashion. Such applications typically involve a large set of observations or measurements, that is queried in a fashion similar to enterprise decision support applications (see the TPC-H benchmark for a sample decision support database [67]). Due to their relatively flat structure, such applications naturally fit into the relational model and can be easily supported by commercial database management systems, with the Sloan Digital Sky Survey (SDSS) [40] being a well-known example. Relational databases already provide storage and query support, so the challenges for this class are in optimizing database organization in order to maximize the

performance benefits of modern query execution engines.

The second category includes applications that process more complicated data structures that are disk-resident due to their volume, but are not effectively supported by existing database methods. High-performance computing applications, for example, involve the processing of large-scale multidimensional *mesh* structures stored on the disk. Such structures cannot be directly stored in existing relational systems, at least without resulting in cumbersome, inefficient implementations. Applications such as high-performance computing and scientific simulation are challenging in that they require the development of *novel* indexing and query processing techniques, that match their complex multidimensional content.

In the following sections, we discuss the specific key challenges in each class, that motivate this dissertation.

### 1.1.1 Relational Database Support for Scientific Applications

Despite the long history of relational database development, using relational technology for scientific data has to address several pressing problems, as scientific applications typically involve complex query processing, massive datasets and frequent, large updates.

#### The Need for Automated Database Design

Query execution performance for large scale databases critically depends on the design of database structures, such as tables, indexes and materialized views. The *database physical design problem* essentially asks for a set of database structures that optimizes the performance of an input query workload, while satisfying given resource constraints. Database design optimization is a difficult problem, as it typically

```
select distinct P.ObjID
from
        photoPrimary P,                     -- P is the primary object
        Neighbors N,                        -- N is the neighbor link
        photoPrimary L                      -- L is the lens candidate of P
where P.ObjID = N.ObjID                     -- N is a neighbor record
        and L.ObjID = N.NeighborObjID       -- L is a neighbor of P
        and P.ObjID < L.ObjID               -- avoid duplicates
        and abs((P.u-P.g)-(L.u-L.g))<0.05   -- L and P have similar spectra.
        and abs((P.g-P.r)-(L.g-L.r))<0.05
        and abs((P.r-P.i)-(L.r-L.i))<0.05
        and abs((P.i-P.z)-(L.i-L.z))<0.05
```

Figure 1.1: An example query from the SDSS database. It computes all the objects within 30 arcseconds of one another that have very similar colors.

involves analyzing large query workloads, consisting of complex SQL statements. In addition, the database execution engine is also complex and the performance of executing a query depends on many factors besides the database design, such as dataset characteristics. Addressing the combination of the above factors requires in-depth understanding of database internals, possessed by a few performance experts and even fewer scientists.

Consider the example of Figure 1.1, depicting an real-world query on the Sloan Digital Sky Survey (SDSS) database. The query joins three tables, looking for astronomical objects within a certain distance from each other, that have similar spectral properties. Naively executing the query by simply scanning and joining the tables will be very slow, as each table contains several terabytes of data. On the other hand, using additional database structures, such as B-Tree indexes with appropriate keys can considerably speed-up the query. For example, building an index on the *ObjID, u, g, r, i, z* columns of the *photoPrimary* relation contains only a subset of *photoPrimary*'s data and thus is much faster to access [2].

Optimizing performance unfortunately gets more complicated than selecting a single index. For instance, there exist other indexes that

[2] *photoPrimary* is actually a logical view, defined over the SDSS PhotoObj relation [40]

might be relevant to the query of Figure 1.1. Consider for example two additional indexes on the *Neighbors* relation, one on attributes *ObjID, NeighborObjID* and one on *NeighborObjID, ObjID*. The two indexes provide two alternative orderings of the *Neighbors* records and enable efficient merge join algorithms. The disadvantage of this scheme is the required storage: *Neighbors* is a very large table and the two indexes require additional storage that is almost equal to the size of the original table[3]. Making an optimal indexing decision for a query requires evaluating a number of indexing alternatives, depending among other factors on the query structure and the available resources.

Databases such as the SDSS have to simultaneously consider thousands of queries [52] and therefore the number of design alternatives increases dramatically. Furthermore, the amount of available resources becomes even more important since, even if we know the optimal indexing solution for each query, there might not be enough space to accommodate all the resulting indexes. Generally, constrained database design problems typically translate into optimization problems that are computationally hard [14].

Additional complexity stems from the fact that indexes are not the only way to improve query performance. In the example of Figure 1.1, significant improvements can be obtained by precomputing the joins (for instance, finding spatially clustered objects with some degree of spectrum similarity) and storing the result in a materialized view. Materialized views have similar behavior to indexes, however they do introduce additional degrees of freedom that a designer can exploit to improve performance. Combined search spaces that consist of multiple design features, such as combinations of indexes and materialized views, are likely to contain better designs, but are more expensive to explore.

[3]This fact was actually experimentally verified in [61].

**Quality and Performance Considerations**

The development of *automated* design tools, that offload the complexity of database design from the human database administrators, is currently a very active research area. Due to their volume and performance requirements, scientific applications motivate the development of novel database design systems, that satisfy the following two main goals.

The first goal is improving the quality of the solutions generated by design algorithms. Ideally, when dealing with large volumes of data, we would like to find physical designs that maximize query performance, without requiring an excessive amount of resources. Satisfying resource constraints such as the available storage space, is a key aspect of physical design. While in a 10GB database greedily allocating twice that space or more for indexes that optimize performance is a reasonable tradeoff, the same does not hold when dealing with a 100TB database. A similar constraint is imposed by the existence of updates in the query workload. In applications such as astronomy, updates take the form of new data that gets appended to the existing. An extreme update workload example comes from the Large Synoptic Survey Telescope, which is expected to download 10TBs of data every night [66]. As new data gets appended, existing indexes and materialized views must also be updated, resulting in additional computation and I/O overheads. Providing close to optimal performance for constrained cases is a critical requirement for scientific applications a nd can only be accomplished through the introduction of more sophisticated optimization algorithms.

The second goal is to improve the running time of automated design algorithms. Execution times are important because applications typically require analyzing massive query logs consisting of thousands of queries. Large workload sizes, as well as the combinatorial explosion

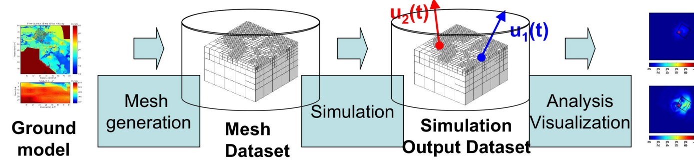


Figure 1.2: An earthquake simulation pipeline.

in the number of considered design alternatives result in long running times and make the design process cumbersome and non-interactive. Improving the scalability of database design tools can not be accomplished simply by reducing the number of design alternatives or queries considered, as such a reduction would interfere with the primary goal of improving solution quality.

Note that developing novel database design tools, although motivated by the extreme requirements of scientific applications, affects a much broader class of applications such as large-scale decision-support and data warehousing applications running on relational DBMS.

### 1.1.2 Support for New Data Organizations

Scientific simulation applications introduce new indexing and query processing requirements. Figure 1.2 shows the architecture of Hercules, a simulation application developed by the Quake group at Carnegie Mellon [4, 70, 47], that computes how earthquakes would propagate for a given ground region and initial conditions. A *mesh* is a discrete model of the ground region under consideration, used by the simulator to compute discrete ground velocity values. The input meshes have typically three-dimensional structures, with explicitly defined topology. The simulation output has the same multidimensional structure, with the addition of a time dimension. Both the input and the output are processed by visualization or general analysis tools.

In modern simulations, mesh models typically consume hundreds

of gigabytes and simulation output volumes are of terabyte scale [4]. Therefore, developing efficient access methods and query processing techniques that can deal with large data volumes is critical for the feasibility of effective simulation applications.

#### Query Processing Requirements

Simulation datasets are primarily processed using multidimensional queries. For example, post-processing and visualization applications compute the value of a simulated parameter (such as the ground velocity for the Hercules example of Figure 1.2) at some random points. Values at random points are obtained through interpolation, using the values computed by the simulation at nearby mesh points. Thus every point that is of interest to an analysis application corresponds to a *point query* on the mesh dataset, that retrieves mesh components that are "nearby" the point. Similarly, a *range query* retrieves a set of mesh components that are contained within a given coordinate range. The typical processing requirements of simulation applications can be mapped to sequences of point and range queries that must be efficiently executed.

Efficiently supporting analysis and visualization applications that perform point and range queries is a demanding problem. Providing high query performance is critical for such applications that require interactive rendering rates of less than 1s per frame [72]. Current applications rely on storing simulation datasets in main memory, using machines (clusters or supercomputers) with sufficient aggregate memory capacity [48]. This solution becomes impractical for terabyte-scale data volumes. What is required instead, is a method for efficiently querying large-scale mesh data stored on the disk.

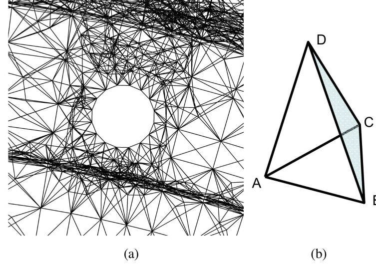


Figure 1.3: (a) Part of a tetrahedral mesh dataset modeling a mechanical component. (b) A tetrahedral (pyramid) mesh element and its four endpoints, the nodes.

**Databases to the Rescue**

Database literature provides a wealth of multidimensional indexing techniques [25]. However, their application to mesh datasets is not straightforward, due to the special structure and geometry of mesh datasets. Figure 6.2 shows an example of an *unstructured tetrahedral mesh*, a mesh organization consisting of pyramid-shaped elements, called *tetrahedra*. Existing techniques do not scale when applied on arbitrarily complex tetrahedral meshes, because the mesh pyramids cannot be effectively captured by simple approximations, like the Minimum Bounding Rectangle (MBR) used in the popular R-Tree index. The approximations typically employed in the database literature cannot effectively address the irregular pyramid shapes, sizes and angles and have suboptimal performance, incurring high storage overhead and construction costs.

Efficiently supporting queries on mesh datasets requires the development of novel multidimensional indexing techniques, that are not affected by the complex geometry of mesh data. An additional requirement is to integrate query processing on meshes to existing database systems as tightly as possible. Tight integration enables applications to exploit advanced performance features on modern databases, such as parallel data access, instead of re-implementing similar functionality. In

addition, integration simplifies application development, since it allows using the same infrastructure for storing both multidimensional and flat data (such as collections of numerical measurements) and combine them efficiently.

## 1.2 Designing for Performance

In this section we outline the contributions of this dissertation in the area of automated database design. While the development of automated design tools has recently sparked intense activity in the database research community, existing tools face limitations in terms of their solution quality and performance. This dissertation develops techniques that overcome the limitations in previous work to derive new, effective and efficient physical design algorithms.

### 1.2.1 Addressing Limitations of Existing Systems

This dissertation improves the state-of-the-art in automated database design by addressing two main limitations of existing approaches. The first is their dependence on cost models that are complicated and inefficient to compute. Modern tools spend more than 90% of their time in evaluating query costs using the query optimizer. Optimizer usage dominates the execution time of design algorithms and renders database design a slow and non-interactive process, especially for large workloads with complicated queries. Furthermore, optimizer usage indirectly affects solution quality, since as only 10% of the available computational resources is spent in actually examining the search space. The time spent in repetitively optimizing the same queries should be better spent in searching for the optimal solution.

The second shortcoming of current approaches is the lack of a formal

framework, that would be amenable to analysis or to the application of standard optimization approaches. Instead, the focus is on the development of ad-hoc, application-specific search heuristics that are geared toward enabling practical implementations: The goal is to reduce execution time by aggressively pruning the search space and by reducing the number of "expensive" calls to the query optimizer. To achieve the above goals, existing algorithms employ a myriad of heuristics, based on simplifying assumptions but with little formal justification.

Currently there exists no analysis (not even an approximate one) on how the myriad of heuristics employed affects the quality of the reached solutions. Consequently, when using existing algorithms to design a system with high performance requirements and stringent resource constraints, like a large scientific database, *the performance of recommended designs could be arbitrarily bad.*

This dissertation addresses the above shortcomings by developing new design tools that provide higher performance and solution quality. First, we develop a novel approach to query cost estimation, that provides the same precision as the query optimizer, but with significantly better performance, thus removing the overhead of query optimization. Second, we introduce a new formal specification for physical design problems, that allows for the development of practical solutions without the uncertainty associated with ad-hoc heuristics. We use our framework to derive physical design tools that are faster and better compared to existing approaches.

An important contribution of this dissertation is an evaluation of the benefits of automated physical design, specifically for scientific applications. We demonstrate how changing the database schema through automated, workload-aware table partitioning improves both query and update performance for large-scale databases. We develop an auto-

mated partitioning tool called *Autopart* [61] and evaluate it using a real-world query workload from the Sloan Digital Sky Survey (SDSS) database, an astronomy application currently deployed using a commercial DBMS.

### 1.2.2 Efficient Use of the Query Optimizer for Automated Database Design

The effectiveness of physical database design tools depends crucially on their ability to efficiently and accurately estimate query costs for a given candidate design. State-of- the-art tools rely on the systems query optimizer for cost estimation: The query optimizer can capture the full complexity of query execution (for instance data distributions) and thus compute query costs with maximum accuracy. Furthermore, since the optimizer has the final decision on whether to use a physical design structure, ignoring the optimizer during the physical design process will most likely lead to designs that will not be used during query execution.

Unfortunately precision comes at a price: Query optimization (an optimization problem by itself!) has a significant cost, as the following simple experiment illustrates: We measured the average time per optimizer call for a commercial index selection tool. The tool was configured to select indexes for a subset of the TPC-H workload and ran on a high-end Xeon 3.0GHz server. Using a server profiler tool, we found that an optimizer call took on average 647ms. At this rate of half a second per call, iterating only once over a set of 100 candidates for a workload of 100 queries would require 1.5 hours!

Our work is motivated by the fact that the query optimizer significantly and unnecessarily limits the scalability of modern automated physical design tools. Reducing the number of expensive optimizer calls during tuning without sacrificing estimation precision will benefit every

aspect of physical design algorithms. The immediate result of eliminating optimizer calls is a reduction in the running time of existing tools by orders of magnitude. Improved performance translates directly to more interactive tuning, which increases the flexibility of the database administrator in exploring more alternative design options. Improved performance also reflects on the size of the problem sizes we are able to handle: Workloads consisting of hundreds or thousands of queries are not uncommon and the only way to handle them currently is through workload compression techniques [3]. Eliminating optimizer calls enables existing database design algorithms to handle much larger workloads, alleviating the need to develop additional workload processing tools, making them relevant only for extreme workload sizes.

Limited scalability due to expensive optimizer calls is also responsible for the wealth of heuristics trying to minimize the number of physical design candidates evaluated by current algorithms. We argue that aggressive pruning heuristics and approximations limit the solution quality achievable (in terms of workload cost improvement) because they restrict the optimization to only a small portion of the overall search space. Intuitively, given the computational complexity of the problem, the more alternatives an algorithm can examine within a given time, the better the chances of reaching a high quality solution.

To minimize the overhead of query optimization, we introduce a novel technique for the efficient *reuse* of optimizer computation during the physical design process. We develop the INdex Usage Model (INUM) a framework that allows us to "cache" previous query optimizer output and reuse it to compute new query estimates "on-the-fly" without further optimizer invocation. INUM is based on the observation that although the index selection algorithm evaluates a large number of candidate configurations, the optimal plan for a given query does not necessarily change from one configuration to the next. INUM captures

exactly the conditions that cause the transition from one optimal plan to the other, when the selection of indexes changes. By eliminating the optimizer overhead, INUM offers significantly improved scalability allowing candidate sets with thousands of indexes and simpler enumeration algorithms.

We implemented and validated INUM with a real commercial DBMS and its optimizer. We present experimental results that demonstrate how the INUM can improve the running time of existing index selection algorithms by orders of magnitude. In addition, for the first time (to our knowledge) we present experimental results from a tuning session involving 100K candidate indexes, a capability way beyond state-of-the-art optimizer based approaches. Our results from an extremely simple enumeration algorithm suggest that considering a large set of candidate indexes can improve recommendation quality by up to 30%.

### 1.2.3 An Integer Linear Programming Approach to Automated Database Design

A major difficulty in the development of practical physical design algorithms is dealing with the huge number of design features, such as indexes or materialized views, that are relevant to an input workload and must be considered. A second difficulty is determining, given a (pruned) search space of candidate features, a combination that provides optimal performance for the workload that also satisfies resource constraints such as available storage. Theoretical studies in index and materialized view selection prove that determining optimal solutions is computationally hard [14].

Existing techniques develop fully engineered tools that involve a multitude of application- specific heuristics. Modern physical design tools select promising candidates by separately analyzing each query in the

workload [15, 2] and traverse the pruned search space in a cost-based, greedy fashion, during which additional candidates can be generated [9]. In addition, to address the overhead of query optimization, several approximation schemes are employed that provide upper or lower bounds to the actual query cost values and are used to guide their heuristic search.

Although heuristics have led to practical implementations, they have an unpredictable impact on solution quality. For example, the greedy search heuristic might be fast, but considering one (or a few) design features at a time ignores the interaction between features and could therefore easily reach suboptimal solutions. By the same token, generating new candidates in parallel with the search is prone to missing important candidates. The above inaccuracies are made worse by cost model approximations, that can easily mis-guide the search into suboptimal paths. There is currently no general analysis on the tradeoff between algorithm performance and solution quality, as the tradeoff is considered a necessary condition for practical tools.

This thesis develops a radically different approach to database physical design. Instead of immediately addressing the engineering of practical algorithms using heuristics we take a step back and model database physical design as a standard combinatorial optimization problem, that must in principle be optimally solved. Our model opens the way for the application of a huge body of work in combinatorial optimization and operations research, that is successfully deployed for real-world, large-scale optimization problems in countless other domains, but not in databases.

We model physical design using an Integer Linear Programming (ILP) formulation, that captures the full complexity of the design problem, while admitting practical implementations. We couple our ILP

abstraction with general,industry strength optimization tools, such as CPLEX, that can find optimal solutions for very large problem instances occurring in practice.

Our ILP formulation offers several advantages to automated design tools both in terms of performance and quality. In terms of performance, using industry-strength optimization engines allows us to process large problem instances, simultaneously considering hundreds of thousands of design alternatives, in less than a minute. The overall performance of our approach also benefits from the integration with the Index Usage Model (INUM).

In terms of quality, our ILP formulation outperforms approaches based on greedy search heuristics. First of all, unlike greedy search, the algorithms employed by ILP solvers are *optimal*. Furthermore, in order to handle large problem instances, our ILP-based approach admits heuristics that improve performance, similarly to existing work. The difference from previous work is that the mathematical structure of the ILP formulation allows us to estimate solution quality loss. Being able to estimate quality degradation, allows us to apply heuristics in a much more informed fashion, avoiding unnecessary simplifications that impact quality beyond a specific threshold. Finally, the quality of our approach indirectly benefits from the boost in performance: By being able to examine more alternatives compared to existing tools, it is unlikely that potentially important alternatives will be “missed”.

### 1.2.4 Workload Driven Schema Partitioning

Relational database systems employ indexes and materialized views to improve query performance. The data volume and query complexity of databases like the Sloan Digital Sky Survey (SDSS) astronomy database [40, 63] make the proper selection of indexes and materialized views

crucial for the effectiveness and usability of the entire system.

Large-scale scientific databases with many users need to make heave use of indexes and views in order to account for the diverse user query workloads. The down-side of the obtained improvements in query performance improvements is increased redundancy. Indexes and views replicate data from the database tables. Therefore they require additional storage and must be updated whenever the base tables are updated. For scientific datasets, the cost of those update operations and the storage overheads can be very high.

We propose table partitioning as an alternative performance optimization for scientific databases. Partitioning directly modifies the database tables and does not introduce redundancy. No additional storage is required and updates do not require additional work. We employ partitioning as the first step in a two-step database design process. The second step consists of designing indexes on the partitioned tables. By first optimizing query performance through partitioning, significantly less indexing effort is required to improve performance.

We developed AutoPart [61],an algorithm for table partitioning based on a representative input workload. AutoPart combines vertical with a form of horizontal partitioning called *categorical* partitioning. We incorporate AutoPart in an automated physical design tool that interfaces to commercial systems, similar already available to index and materialized view selection tools [2, 15, 16, 32].

We experiment with AutoPart in the context of the SDSS database [], running on SQL Server 2000. We use SQL Server's Index Tuning Wizard to design indexes on top of the partitioned schema and we compare the performance of the SDSS workload between the original and the partitioned schemas. Our experimental results demonstrate the benefits of partitioning: In the context of indexing and update

statements, partitioning improves query performance by 20%, while at the same time achieving a five-fold improvement in update performance and requiring half the index space.

## 1.3 Efficient Query Processing for Simulations

Supporting large-scale simulation applications, requires the development of efficient indexing techniques for new data organizations. Simulation applications typically represent application domains by *unstructured tetrahedral meshes.*

Tetrahedral meshes are powerful modeling tools, as they can have varying resolutions and can model arbitrary shapes with high precision. The fact that meshes consist of elements with highly variables shapes, angles and sizes, makes tetrahedral meshes very popular for simulation applications, but also very difficult to index using techniques based on geometric approximations.

The processing performed on tetrahedral mesh datasets by applications such as visualization can be modeled as generic point and range queries. A point query asks for the mesh element containing an arbitrary input point, while a range query asks for the elements contained within an input rectangular range, or the elements intersected by it. Point queries are used primarily for interpolation, as the information stored at the nodes of the containing element are used to compute simulated values at arbitrary points within the tetrahedron. Range queries have general applicability, for instance for selecting and loading a region of elements in main memory.

Database literature provides a wealth of spatial indexing techniques [25]. Existing techniques do not scale when applied on arbitrarily complex tetrahedral meshes, because the pyramids cannot be effectively

captured by geometric approximations like the Minimum Bounding Rectangle (MBR). Approximations using more complex, regular shapes can still not fully capture arbitrarily shaped tetrahedra. Such techniques typically have also complex implementations, incurring unnecessary storage and construction overheads. Furthermore, specialized multidimensional indexing techniques are not integrated to existing database management systems and therefore can not take advantage of features such as standardized interfaces, efficient low-level storage management and parallel data access, without implementing such functionality from scratch.

We develop Directed Local Search (DLS), an efficient query processing technique for point and range queries on unstructured tetrahedral meshes. DLS avoids the complexity involved in capturing mesh geometry, by utilizing the connectivity between mesh elements. It uses a novel application of the Hilbert curve to obtain an initial approximate solution, which is then "refined" through searching *locally*, in the vicinity of the initial solution. Our technique relies on the distance preserving properties of the Hilbert curve and on an efficient representation of connectivity information to provide significantly better performance compared to traditional techniques that rely only on geometric approximation. Furthermore, DLS does not require the development of radically new access method. It is based on B-Trees and can therefore be easily integrated with existing DBMS and take advantage of the features and efficient implementations they provide.

We extend the idea of relying on mesh topology for query processing, by using it to cluster mesh elements on the disk. Clustering refers to arranging mesh elements on disk pages, so that "regions" of connected elements get retrieved with the minimal number of page accesses. A region in a mesh can be defined either through a rectangular range query, or can have a more general, arbitrary shape. We call the latter shapes

*features* and the corresponding queries *feature queries*. In the earthquake analysis example, a feature query might ask for all the elements that belong to the ground surface. The retrieval performance for a set of mesh features depends on the way the elements that belong to the queried features are arranged on disk pages.

Previous approaches for clustering multidimensional data on the disk rely on space-filling curves. While space-filling curves perform well for rectangular (and particularly cube)-shaped queries, they are not suited for more general feature queries. We propose a novel layout technique for feature queries, that is based on graph partitioning. We model the mesh as a graph, with the "neighbor" relationships between elements corresponding to graph edges and use a graph partitioning algorithm to distribute the graph nodes ( the mesh elements, that is) in page-sized partitions. We demonstrate that with a suitable assignment of edge weights to the graph, graph partitioning outperforms space-filling curves for arbitrary feature queries.

# Chapter 2

# Related Work

This chapter presents related work, broadly covering three major research areas:

1. Systems for large-scale scientific data management.

2. Automated database physical design and self-tuning architectures

3. Multidimensional indexing.

## 2.1 Systems for Large-Scale Scientific Data Management

In this section we review prevailing approaches in managing large scale scientific datasets. We roughly classify them into those using DBMS technology and those based on (indexed) file organizations.

Although DBMS-based approaches are more recent and not yet widely accepted, there is an ongoing effort to integrate them with traditional file-based techniques, combining the best of both worlds [28, 18].

## 2.2 Managing Scientific Data using a DBMS

The first encouraging results from the application of DBMS technology in scientific data management come from Jim Gray's pioneering work on the Sloan Digital Sky Survey (SDSS) database [40, 63]. The SDSS is the largest astronomy survey ever undertaken, aiming to create a three-dimensional map of a quarter of the sky. It stores information for hundreds of millions of astronomical objects and is expected to reach the size of 15TB of formatted information. The SDSS data is stored in the *SkyServer*, a system using Microsoft's SQL Server 2005 relational DBMS for data storage and Microsoft's web services infrastructure for making data available for querying on the web [1]. Queries can be submitted to the system through SQL, form-based an graphical user interfaces.

The SDSS deployment demonstrates two major advantages of using relational DBMS technology over the traditional file-based approach:

1. *Ease of application development and portability.* Traditional file-based approaches dictate the development of application-specific tools for extracting and processing data. Developing such tools requires that scientists/astronomers possess programming skills and is a slow and cumbersome process. Furthermore, the resulting tools are tied to specific data formats, which are often poorly documented and are difficult to distribute and share.

   Using a relational DBMS, on the other hand, allows scientists to express their data processing requirements in a declarative fashion, using SQL. Developing SQL queries requires some initial training, however it has been observed that it considerably reduces application development time. The reason is that SQL allows users to

[1] http://cas.sdss.org/dr5/en

focus on specifying the the information they are interested in at a high level, without concerning themselves with implementation details, such as file access, memory management, algorithms etc. Furthermore, using SQL helps in creating portable applications because, as long as the database specification remains the same, the applications are independent of how the database back-end is implemented.

2. *Performance* There is strong indication that DBMS performance features, currently primarily used to manage hundreds of terabytes of commercial data, is directly applicable to scientific applications [28]. Using a DBMS has the advantage that the database system and not the scientist's code, is responsible for performance. Modern DBMS offer a variety of ways to efficiently query large volumes of data, such as optimized and flexible indexing, automatic inter-query parallelism and parallel I/O and facilities such as distributed query processing and replication. Such facilities, while nearly impossible to incorporate into custom applications, at least without reinventing the wheel, are readily available to the scientists through using a standard commercial DBMS.

The performance and usability advantages of modern DBMS technology and the successful *SkyServer* deployment has sparked a number of follow-up research projects. A major challenge is extending the existing relational DBMS infrastructure to support complex data types, such as spatial and multidimensional data. The motivation for this work is that, while most existing DBMS do not directly provide multidimensional access methods, such structures can be "simulated" by smartly combining B-Tree indexes and custom application code, implemented as external libraries. The reasoning behind this approach is that using existing structures exploits the DBMS facilities for data

storage and query processing.

[29] develops the Hierarchical Triangular Mesh (HTM) index, a regular decomposition technique for querying spatial (astronomy) data organized in spherical coordinates. [20] presents implementations of multidimensional indexes and search algorithms, such as nearest neighbor search in a traditional relational DBMS settings and reviews case studies using SDSS data and SQL Server. [51] reports on a galaxy clustering application using DBMS and Grid technologies, that is an order of magnitude faster compared to traditional, scripting-based methodologies. Finally [52] develops a system for managing user queries and system resources in large scale astronomy databases.

Besides astronomy, recent work uses DBMS technology to support Finite Element Analysis (FEA) applications [35, 36, 37]. The authors argue that DBMS technology is advantageous in supporting FEA workflows, compared to more traditional file-based approaches. First, they use the expressive power of the relational model to specify a *logical* schema of the data involved in an FEA application, that can be easily documented and shared with users. A key aspect is also implementation-independence: The schema and thus the applications that use it do not need to change when the physical data storage changes. Even changes to the schema (schema evolution) can be relatively easily masked through *logical views* if necessary. Given a logical data specification, visualization, analysis and sharing tools can be layered on top of the database, using SQL statements to extract data. The applications do not need to worry about physical access to the data or performance: the DBMS implements physical access transparently, using indexing and parallel I/O and query processing for performance.

## 2.3 Non-DBMS Architectures

The work on *Computational Databases* [70, 68, 69] uses database access methods to support scientific computing applications with massive (terabyte-scale) datasets. Their goal is to support mesh management tasks, such as mesh generation and partitioning, for large meshes comprising more than a billion of elements. Traditional simulation architectures that are based on main-memory processing do not scale for large mesh sizes of tens or hundreds of gigabytes, as standard workstations or even small clusters do not have the necessary physical memory. The computational database approach is to move mesh generation to the disk, using database techniques and specialized processing algorithms. The proposed disk-resident approach is currently used to support large scale earthquake modeling and simulation [4].

Despite the success of Computational Databases and of the DBMS-based approaches described in Section 2.2, database techniques are not yet widely adopted. The majority of scientific applications stores their data in files, formatted according to specifications such as NetCDF [56] or HDF [64]. Existing scientific data formats provide standardized representations and support for basic data types such as integers, floats, strings and multidimensional arrays. Their purpose is to provide standardized and efficient storage for structured data that are not efficiently supported by relational databases, such as multidimensional arrays and time series [65].

File-based approaches lack standardized indexing mechanisms, as indexing is performed with custom, application specific code. Furthermore, modern scientific applications need to process information scattered over millions of files, a requirement nearly impossible to efficiently support given the scalability and metadata capabilities of current filesystems, unless of course filesystems become databases! [28]. To

address the first disadvantage, several projects are underway to provide general indexing facilities for HDF file formats [24].

A more comprehensive approach is to integrate scientific data formats with the database. Integration is enabled by new database features that allow storing and retrieving large binary objects as part of a database record (at least conceptually), or use actual pointers to the file system. Such objects are opaque to the DBMS and can be used to store entire HDF files or parts thereof. Another key capability of modern DBMS in integration with user defined code: This allows the DBMS to incorporate specialized accessed methods, if necessary. Integrating scientific file formats with the database allows for "hybrid" systems that will be combine the advantages of both approaches. Exploring the combined design space is still at its first steps [28, 18].

## 2.4 Automated Database Design

This section discusses related work in automated physical design and self-tuning databases. We discuss the main approaches proposed in the literature for selecting indexes, materialized views and for performing table partitioning. We then present related work on other related database configuration problems.

### 2.4.1 Index Selection

The problem of index selection in relational databases was the first physical design problem to be comprehensively studied by the database community and to lead to actual commercial implementations. The index selection problem is defined as follows (adapting the specification from [14]):

*"Given a workload $W$ consisting of SQL statements and a storage space*

*of S bytes, determine a set of indexes that minimizes the cost of executing the statements in W and requires a total storage that is less than S.”*

The work in [15, 32] introduces the basic approach to index selection. The ideas in [15, 32] are important, as they form the foundation for the physical design algorithms currently employed in commercial systems.

Although the systems in [15, 32] were developed independently, they are similar in that they introduce three distinct components for index selection systems. *Candidate Selection* computes a set of “candidate” indexes, that the algorithm will examine. The *Search* module searches through the space of possible index combinations, to find the subset of candidates that minimizes workload cost. Finally, *Cost Estimation* is responsible for estimating workload costs under different index subsets, so that the optimal solution can be identified. We next describe the three components in more detail.

#### Candidate Selection

The goal of candidate selection is to perform an initial pre-selection of indexes, that are “candidates” for inclusion in the final design. Preselection is necessary as the unrestricted set of all the indexes that are potentially relevant to the workload could be very large. For instance, the number of all possible indexes on a table with $n$ attributes is given by the following formula [32]

$$\sum_{k=1}^{n} 2^k \times \frac{n!}{(n-k)!} \tag{2.1}$$

Candidate selection is typically performed in a *per-query* fashion. One approach is to determine the optimal set of indexes for each query individually [15] and compose the candidate set by joining the resulting “per-query” optimal indexes. Optimizing each query individually is

an easier task, as the number of indexes that are relevant to a single query is much smaller compared to the number of indexes that are relevant for the entire workload. [15] optimizes each query individually by trying out a set of different index combinations (computed through syntactic analysis) and selecting the best. [32] first computes a number of syntactically relevant indexes per query and submits all of them to the query optimizer. The plan selected by the optimizer will by definition use the optimal set of indexes.

#### Search

Once a set of candidates is determined, the next step is to compute the subset that minimizes workload costs. The problem can be formally specified as follows:

*"Given a set $I = I_1, I_2, ..., I_k$ of candidates, select a set $I_{min} \subset I$ that minimizes the cost of workload $W$ subject to $\sum_{i \in I_{min}} storage(i) \leq S$".*

As it is impractical to exhaustively enumerate all possible subsets of $I$, both approaches fall back to simple search heuristics. The $greedy(m, k)$ approach of [15] forms a solution in an incremental fashion, by performing several passes over the candidate set $I$ and returning a subset consisting of at most $k$ indexes. The solution is formed by first exhaustively computing the best possible subset of $m$ indexes. This "seed" is extended by adding indexes in a greedy fashion, picking at each step the index that provides the largest improvement in workload performance compared to the current state.

An alternative approach is to exploit the similarity of the index selection problem to the 0-1 knapsack problem [32]. The algorithm in [32] computes for each index in $I$ its benefit, defined as the improvement in estimated execution time that an index contributes to all queries that exploit it. Then, the indexes are sorted in a decreasing ratio of benefit

to size and are added to the solution until the storage constrained has been reached.

The limitations of the proposed search strategies are also described in the literature. The greedy search theoretically can produce arbitrarily bad results [15]. In addition, considering one index at a time ignores cases where indexes benefit workload performance only when selected together. Ignoring *index interactions* can cause the greedy search to miss useful indexes and thus form suboptimal solutions. This is why the $greedy(m, k)$ involves a small exhaustive selection phase. To control execution time, the value of $m$ is relatively small, 1 or 2.

Knapsack-based approaches [32] lack theoretical justification, as the index selection problem can not correctly be mapped to a knapsack problem. The problem is that while in the knapsack formulation, the benefit of each object under consideration is well-defined, the benefit of an index also depends on the other selected indexes, due to the same index interaction effect, and cannot be independently assigned to indexes. Therefore, there are no guarantees about how close the knapsack heuristic can get to the optimal solution. [14] presents a heuristic for reassigning benefit values to indexes, that can in some cases help to compute per-instance quality guarantees.

#### Cost Models

An important component of automated design tools is their *cost model*, the method they use to estimate the workload cost for the various sets of indexes under consideration. The ideal way to estimate workload performance for a given set of indexes is to implement the indexes in the database, load them with data, execute the queries and measure the improvements in query execution time. This is an impractical procedure, since ideally we would like to try out hundreds or thousands

different index sets before making a choice

The alternative adopted by automated design tools is to *estimate* workload costs using the cost estimation facilities provided by the query optimizer [58]. The query optimizer is the most accurate way to estimate workload costs, as it is designed to take into account details such as the costs of various operators and the effect of data distributions. In order to avoid actually building the indexes under consideration, index selection tools introduce “what-if” interfaces [15], that “simulate” the existence of indexes simply by making their statistical information available to the query optimizer.

Although using the query optimizer is more efficient than actually implementing the indexes under consideration, query optimization is still a very expensive process, especially since the optimizer is continuously invoked by the index selection tool. To minimize the number of calls to the optimizer, index selection tools use additional heuristics to compute *approximate* query costs [15]. Specifically, proposed techniques approximate the cost of a query $Q$ for a set of indexes $I_1$ using previously computed cost values for $Q$, under a different index set $I_2$.

#### Extensions

We now present follow-up work, aiming to improve the basic architecture described in the previous sections. One limitation of the basic framework is the limited number of candidates selected by the per-query-optimal approaches of [15, 32]. Consider an index that improves performance for *all* the queries in the workload, but is not optimal for any of the queries. This index will not be included in the candidate set, although it could be part of the optimal solution. *Index merging* [16] is a strategy for extending the per-query index set by combining (merging) candidates. Merging generates new candidates that can still

improve query costs, although they are not optimal for any query. The advantage of merged candidates is that they make better use of the space, as the same index is usable by several workload queries.

Merging is only one of many ways to extend a set of candidate indexes. [9] introduces several index transformations beyond merging. For example, the *prefixing* transformation generates, from index $I$, a thinner index $I_{prefix}$ containing only the $k$ first columns of $I$. The thinner index might perform worse than the original, however it occupies less space and thus might still be useful.

Blindly using transformations is likely to cause an explosion in the number of candidates, that are output by the candidate selection phase. [9] proposes a combination of candidate selection and greedy search. At each step in the search, the already selected partial solution is extended by considering several transformation and selecting the best. The transformed solution becomes the starting point for the next step. The transformations in [9] are designed to always reduce the storage requirements or the update costs of a given set of indexes. Therefore the algorithm is guaranteed to terminate after a finite amount of transformations, as eventually the problem constraints will be satisfied by the reached solution.

### 2.4.2 Materialized Views and Partitioning Selection

After indexes, the next step in the development of automated database design tools was to add support for materialized view selection. [2, 73]. In terms of physical design algorithms, materialized views are very similar to indexes. A set of candidate views can be selected from the workload, first by determining per-query optimal views and then by augmenting the view set through index merging. The resulting set of candidate views is added to the set of candidate indexes and the *joint*

search space is explored by a search strategy similar to those developed for indexes[2].

Supporting multiple design features in an integrated fashion, as described above, generates very large search spaces, increasing the number of solutions that must be examined. A solution to this problem would be to select multiple features in stages, with each stage focusing on a single feature. For example, the *INDFIRST* strategy of [2] first computes a solution consisting only of indexes and then augments this solution with materialized views. Staged approaches are attractive because they naturally restrict search spaces and are more modular, because different algorithms can be used for different features.

Staging has the disadvantage of ignoring interactions between indexes and views: cases where a combination of one or more indexes or views is more beneficial compared to its members considered in isolation. Since combinations of different features are not considered in a staged approach, it is possible that opportunities for positive interactions might be lost. [73] proposes a hybrid approach, where both staged (called "iterative" in [73]) and integrated algorithms co-exist. [73] presents an analysis of physical design features, classifying their pair-wise interactions as "strong" or "weak". Intuitively, strong interactions require integrated algorithms, while weak interactions admit staged solutions without significant penalties.

[3] adds vertical and horizontal partitioning to the physical design tool described in [2]. The main focus is on extending the basic approach of per-query candidate selection and merging to handle partitioning. The unavoidable increase in search space size is avoided by integrating candidate selection with the search: In a manner similar to [9], the partial solution computed after a search step is used to compute more

[2]The candidate set is also extended by a set of indexes *on* the materialized views.

candidates to be evaluated by the subsequent search step.

### 2.4.3 Other Research on Physical Design

Besides the "traditional" database physical design problems described above, there is significant work on other aspects of database design, dealing with different systems, physical design structures or design scenaria.

[55] describes an approach for workload-aware partitioning database tables in the context of a parallel DB2 database. The difference from single-node database design is that in a parallel system, multiple nodes are involved in query execution. Each node can process local or remote data, depending on how the data is partitioned or assigned to nodes. The data allocation and the query plans determines the number of nodes participating in query execution along and the potential communication costs, which are parameters that did not exist in the single-node case. The goal is to determine a table partitioning method (based on key ranges, key hashing, or even replication) that optimizes the performance of a given query workload. Despite the slightly different specification, the approach followed to solve the problem is similar to the single-node design algorithms. Query execution costs are modeled using the query optimizer, that is also responsible for recommending a number of partitions on a per-query basis. This initial population of partitions is then processed using transformation heuristics (including a genetic algorithm) to derive a design with a minimal cost. The optimizer is used in every step to evaluate different design alternatives. [46] uses a similar approach to determine a set of clustering attributes in DB2's Multi-Dimensional Clustering Tables (MDC).

A problem related to automated database design is that of dealing with changes in the workload or the data. Ideally, we would like

the database to monitor the query workload and adapt the existing database design to workload changes. [10] presents a “database design alerter”, a system whose goal is to monitor an incoming query workload for new queries that might not perform well with the existing physical design. In the event of such workload changes, the system can either alert the database administrator, so that a full-blown design session is executed, or recommend changes providing upper bounds for the achievable performance. The focus of the paper is on efficiency: The tool should identify queries that would benefit from changes to the current physical design while the system is in normal operation. Therefore, the alerter should refrain from using the optimizer for cost estimation and consume minimal processing resources.

[11] follows a different design approach, where the focus is on balancing index creation costs and query execution costs when the future queries are unknown. The idea is that constructing an index represents an investment in system resources, that is amortized through the efficient execution of multiple queries that use the index. Generating (and dropping) highly specialized indexes very frequently is potentially expensive, on the other hand conservative index creation results in suboptimal query execution. The authors model the problem as an online problem, seeking an index selection strategy that bounds the performance losses introduced by the lack of knowledge about future queries. The idea is to execute an index after observing $x$ queries that could potentially use it, where $x$ depends on the relative index creation and query costs. They provide an “idealized” 3-competitive algorithm, however, due to implementation issues their approach deteriorates to a simple “reactive” mechanism, with additional heuristics for preventing phenomena such as “oscillations”.

### 2.4.4 Theoretical Approaches

Not surprisingly, automated design problems have also attracted theoretical attention, since they typically translate to optimization problems. [17] shows that under certain assumptions, the materialized view selection problem has an exponential lower bound in the number of alternative materialized views that must be examined. [34, 42] deal with the problem of building materialized views on data cubes and prove that the problem is NP-hard.

[38] deals with the problem of automated index selection. It describes an approach based on Integer Linear Programming (ILP), making the assumption that a query can use only a single index and that the cost of accessing and index and its storage cost are related. To solve their resulting ILPs, they introduce a randomized rounding algorithm that achieves optimal workload performance with high probability, with a near-optimal storage requirement. Their approach however is not applicable to index selection scenaria in real database management systems, as multi-table queries can typically use more than one indexes, the query cost is not directly related to the index storage and there also exist update costs.

The work described in Chapter 4 of this dissertation is very similar to [12]. Their ILP formulation accounts for queries using more than one indexes and also models index update costs. They also provide a specialized branch-and-bound procedure for its solution. However, they ignore the existence of storage constraints and thus it is not clear whether their branch and bound procedure is valid. Our work is based on the same ILP formalism, but is concerned with applying it to real-world problems, that involve commercial systems and databases. Specifically, we resolve issues related to generating an ILP instance from a real database and workload, where it is impossible to capture the full

complexity of the problem and approximation is a necessity. We also provide solutions to the problem of efficiently computing query costs and characterize the effect of query cost approximation. Finally, we rely on commercial ILP solvers and compare our ILP-based systems with existing, real-life index selection tools on commercial database systems and workloads.

## 2.5 Multidimensional Indexing

Existing techniques for multidimensional indexing become relevant for the part of this dissertation dealing with indexing scientific data and in particular with the problem of dealing unstructured tetrahedral meshes (Chapter 6). Database literature in fact provides a wealth of multidimensional indexing techniques: An excellent survey is [25].

R-Tree based approaches approximate objects by their Minimum Bounding Rectangles (MBRs) and index them with an R-Tree [31] variant. The R-Tree search for a query starts from the root level and follows a path of internal nodes whose MBR intersects the query range or contains the query point. If multiple nodes at one level match the query criteria the search will follow all possible paths, requiring more page accesses. There exists a large body of research on improving performance by minimizing the area of R-Tree nodes and the overlaps that lead to multiple paths. Dynamic techniques like the original R-Tree construction algorithm [31] and the R* tree [6] maintain an optimized tree structure in the presence of data updates. Static techniques like the Hilbert-packed R-Tree [41], the Priority R-Tree [5] and others [57, 21] attempt to compute an optimal R-Tree organization for datasets that do not change.

Extensions like the P-Tree [39] attempt to improve R-Tree perfor-

mance by using Minimum Bounding Polyhedra. The P-Tree relies on polyhedra with faces aligned to a fixed set of $d$ orientations. Such 'constrained' polyhedra will likely still lead to overlaps, as they do not exactly capture a tetrahedral element. Also, more sophisticated bounding elements require more storage and reduce the tree fan-out (As pointed out also in [25], the study on P-Trees [39] suggests that 10 dimensions are required for efficiently storing two-dimensional objects with arbitrary orientations). Finally, computing the polyhedra orientations that have optimal performance for a given dataset requires complicated preprocessing [8].

A clever solution to the problem of overlaps is to use non-overlapping regions. The R+ Tree [59] ensures non-overlapping tree nodes by generating disjoint MBRs during node-splitting and *replicating* the objects that cross MBR boundaries. The search algorithm then accesses only useful nodes, those that really contain a part of the query reply. On the other hand, the set of nodes retrieved will contain multiple pointers to the same objects, wasting I/O bandwidth and requiring additional post-processing. The R+ Tree trades query performance for higher storage requirements and higher construction time. The construction complexity is higher, because a single inserted object might result in a large number of replicated object insertions and essentially a large number of random I/O operations.

More sophisticated clipping-based techniques like the cell-tree [30] partition the indexed space into disjoint convex polygons. This solution is ineffective for tetrahedral mesh datasets for two reasons. First, the cell-tree construction algorithm does not preclude objects crossing partition boundaries that have to be replicated, like in the R+ Tree case. Furthermore, the space overhead of keeping the polygon descriptions in the tree nodes is much higher compared to that of storing MBRs and leads to poor storage utilization for large datasets.

Another approach is to overlay a rectilinear grid over the indexed domain and approximate each object by one or more grid cells. The cells are arranged and indexed using coordinate transformations like the Z-order [53, 54].

Finally, another family of techniques translates MBRs into points of a higher dimensional space and uses a point access method to index them [25]. Transformation-based approaches have the well known disadvantages of transforming point queries into open-ended range queries, of decreasing the uniformity of the indexed domain and of destroying spatial locality. Furthermore, since multiple MBRs overlap with the query region (or contain the query point), those techniques retrieve a potentially large superset of the query answer, which, after paying a high I/O retrieval cost, must be post-processed to obtain the final answer [25]

# Chapter 3

# Efficient Use of the Query Optimizer in Automated Database Design

## 3.1 Introduction

Algorithms for automated design and performance tuning for databases are gaining importance, as database applications are getting larger and more complex. 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 resource constraints, such as the available storage. Design tasks typically translate to hard optimization problems for which no efficient exact algorithms exist [14] and therefore current state-of-the-art design tools employ heuristics to search the design space.

Figure 3.1 outlines the two-stage approach typically employed by database design tools [2, 15, 32, 73]. 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

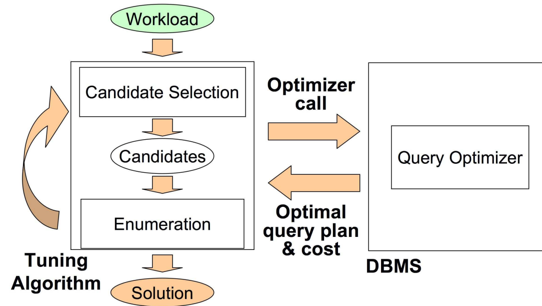


Figure 3.1: Database design tool architecture.

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 very 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 [9]: "...we would require hundreds of optimizer's calls per iteration, which becomes prohibitively expensive". Our experiments show that up to 90% of the running time of an index selection algorithm is spent in the query optimizer.

*This dissertation presents a way to minimize the overhead of query optimization for automated design tools, without sacrificing precision.* We present the INdex Usage Model (INUM), a framework for caching and reusing optimizer results. The intuition behind the INUM is that although design tools must examine a large number of alternative designs, the number of different optimal query execution plans and thus the range of different optimizer outputs is much smaller. Therefore it makes sense to *reuse* the optimizer output, instead of repeatedly computing the same plan.

The INUM works by first performing a small number of key optimizer calls per query in a *precomputation* phase and caching the optimizer output (query plans along with statistics and costs for the individual operators). During normal operation, query costs are *derived* exclusively from the precomputed information without any further optimizer invocation. The derivation involves a simple calculation (similarly to computing the value of an analytical model) and thus is significantly faster compared to the complex query optimization code.

Our experiments with INUM using a commercial optimizer and real workloads show that it provides three orders of magnitude faster query cost estimation for index selection algorithms, while the returned cost estimates are equal or in the worst case very close to those that would have been returned by an actual optimizer call.

We present our solution to cost estimation for physical design in the context of index selection algorithms, however it is important to note that our approach is also applicable to other physical design features (e.g. materialized views, table partitions).

### 3.1.1 Contributions

Our work makes several contributions to the area of automated index selection algorithms:

1. Improved running time for existing index selection tools. The 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. Our experiments demonstrate that an index selection algorithm with INUM provides three orders of magnitude faster cost estimation. When factoring in the precomputation phase that in-

volves the optimizer, we measured execution time improvements of 1.3x to 4x and this without implementing any of the techniques proposed in the literature for optimizing the number of cost estimation calls. Another aspect of faster cost estimation is that it allows an INUM-enabled tool to process larger workloads compared to an optimizer-based tool, in the same time.

2. Better scalability in terms of the number of candidate indexes examined. Existing candidate selection heuristics aggressively prune the space of candidates in an effort to maintain reasonable execution times. INUM allows existing search algorithms, such as the greedy search of [15, 2] to examine 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.

   Using INUM we were able to evaluate 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 a couple of hours, a prohibitively expensive task for existing optimizer-based tools. The solution derived from this "'power" test improved the solution given by a commercial tool by up to 20%.

3. 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 [9] generates thousands of new candidates combinations, a number that makes optimizer evaluation prohibitively expensive. The INUM can replace the approximation logic currently used [9], allowing the algorithm to use exact query costs, as opposed to upper bounds, thereby avoiding estimation errors and the corre-

sponding quality degradation.

4. Improved performance for online design tools. Online monitoring [10] and tuning [11] tools respond to changes in the workload or the database by alerting the administrator or automatically adapting the design. Online algorithms run frequently and during normal operation, therefore it is critical that the resource-intensive query optimizer is used as rarely as as possible. INUM's cost estimation is lightweight (since the optimizer is not involved after the initial precomputation phase) and accurate, improving on existing techniques that are based on upper-bounding query costs using "locally-optimal" plans [10].

   Online environments are a particularly good match for INUM's plan caching framework because, unless the workload changes dramatically, a large fraction of the cached computation will be reusable among multiple sessions, amortizing the initial setup cost.

The rest of the chapter is organized as follows: Sections 3.2 and 3.3 present the foundation for the Index Usage Model. Sections 3.4 and 3.5 describe our algorithms for caching and reusing query execution plans. Section 3.6 presents the extensions to basic INUM that are essential for handling the full complexity cost estimation. Sections 3.7 and 3.8 present our experimental results and Section 5.7 concludes.

## 3.2 INUM Fundamentals

We show how the INUM accurately computes query costs while at the same time eliminating all (but one) optimizer calls, under certain restrictive assumptions on the indexes input to the INUM. This section sets the stage for the complete description of the 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 $W$. Let $Q$ be a select-project-join query accessing 3 tables ($T_1$,...,$T_3$). 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 $x \leq a_{T_i} \leq y$.

During both the candidate selection and enumeration phases, the 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 "configuration" to denote a set of indexes, according to the terminology in previous studies [15].

We require that the configurations submitted by the tool to the optimizer have the following two properties:

1. *They are restricted to 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*.

2. They are *atomic*. We use the notion of an *atomic* configuration exactly as previously used [15]:

**Definition 3.2.1** *A configuration $C$ is* atomic *with respect to query $Q$ if there is a possible query execution plan that uses* all *the indexes in $C$.*

We assume the configurations submitted to the optimizer and to the INUM involve at most one index per table and have the form: ($T_1$:

$I_{T_1}$, $T_2$: $I_{T_2}$, $T_3$: $I_{T_3}$) where any of the $I_{T_i}$ values can be empty. Query costs for any configuration $C$ can be derived from its constituent atomic configurations [15] and current index selection tools take advantage of this fact by submitting only atomic configurations to the optimizer. Notice that we do not handle atomic configurations with more than one index per table. Our techniques can be extended to handle this case by modeling the *index intersection* operator. Such extensions are part of our ongoing work.

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 [9]) to reduce the number of optimizer evaluations. In the next sections we show how through reasoning on optimizer operation we can obtain accurate query cost estimates efficiently, without any query optimization overhead.

### 3.2.2 Reasoning About Optimizer Output

Assume we have already performed a single optimizer call for query $Q$ and a configuration $c_1$ and obtained an optimal plan $p_1$. We show that we can *reuse* the information in plan $p_1$ to compute $Q$'s cost for any other configuration $c_2$ that satisfies the non-join column restriction from the previous section. We precisely define *plan reuse* using the fol-

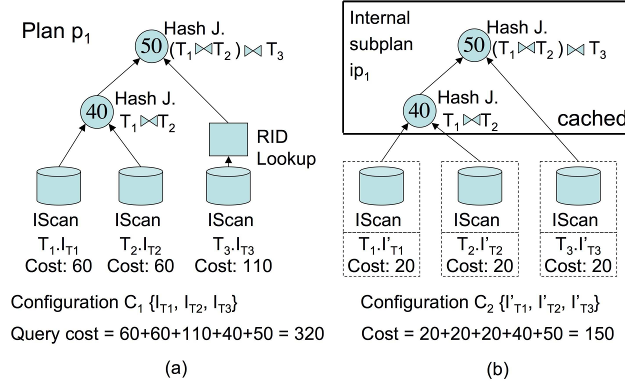


Figure 3.2: Illustration of plan reuse. (a) The optimal plan $p_1$ for configuration $C_1$. (b) The cost for $C_2$ is computed by reusing the cached internal nodes of plan $p_1$ and adding the costs of the new index access operators, under the assumptions of Section 3.2.

lowing algorithm.

**Input:** Query $Q$, plan $p_1$ from the optimizer (for configuration $c_1$), new configuration $c_2$.
**Output:** New cost for query $Q$ for configuration $c_2$.

1. 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.

2. For every table $T_i$, construct the appropriate data access operator (index scan or seek with optional RID lookup) for the corresponding index in $c_2$ (or a table scan operator if the index does not exist). Add the data access operator to $ip_1$ in the appropriate position.

3. $Q$'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

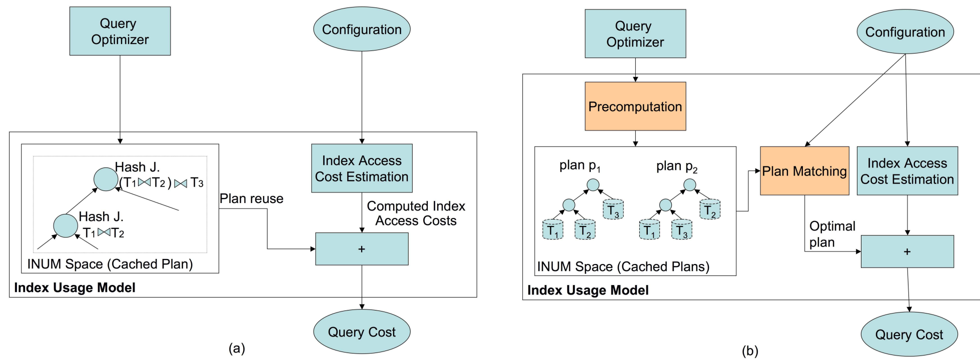


Figure 3.3: (a) Cost estimation with INUM for the example of Section 3.2. (b) Complete INUM architecture.

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 dramatically improves performance. 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.

If the internal structure of plan $p_1$ (the subplan $ip_1$) is optimal for $c_2$, then the cost returned by the reuse procedure is equal to the cost that would have been returned by the optimizer.. For our particular scenario, we prove that there exists a single optimal subplan for $Q$, regardless of the configuration $c_2$, as long as all the non-join column constraint of Section 3.2.1 is satisfied. Thus one invocation of the optimizer is enough to obtain the optimal plan $p_1$, which can then be reused according to our reuse procedure. (Note that the reuse procedure will

yield incorrect results, when used with configurations that violate the non-join column restriction). 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. 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 $cost(ip_1)$, $cost(ip_2)$ their total costs, with $cost(ip_1) < cost(ip_2)$.

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, 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 $cost(ip_1) < cost(ip_2)$ 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 3.2.2** *An interesting order is a tuple ordering specified by the columns in a query's join, group-by or order-by clause [58].*

**Definition 3.2.3** *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.*

We can extend the approach presented in this section to queries involving aggregation, group-by and order-by clauses. Theorem 3.2.1 applies, as long as the input configurations do not cover any of the interesting orders in the query. Intuitively, this non-interesting order restriction implies that no configuration is fundamentally more "powerful" than the others and therefore there is no reason for more than one optimal plans.

Figure 5.8 (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 (which could be implemented with analytical formulas or the appropriate optimizer modules) to estimate the cost of accessing the indexes in $C$. The sum of the index access costs for $C$ is added to the cost of the internal subplan to obtain the final query cost.

## 3.3 INUM Overview

Unlike the scenario of Section 3.2, in a real index selection session the "no join column" restriction is invalid. In fact, join columns are among the primary candidates for indexing. 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 plans for a given query.

To see why configurations that cover interesting orders lead to more than one optimal plans, consider the example query of Section 3.2 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 on $T_2$ and $T_3$. The optimal plan in this case could be different, first joining $T_2$ and $T_3$ with a merge join.

Figure 5.8 (b) shows the architecture of INUM used to address the general case. The INUM takes as input requests from an index selection

tool consisting of a query and an atomic configuration for evaluation. The output is the optimal plan for the query under this configuration and the query cost.

The *INUM Space* contains for each query the set of precomputed plans that are used to derive the query costs. In the simplified example of Section 3.2, the INUM Space contained only a single plan. If we remove the restrictions discussed in Section 3.2 the INUM Space is likely to contain more plans per query. 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.

The *INUM Space* is a central component of INUM. It is a cache of *all* the plans (per query) that could potentially be returned by the optimizer as optimal for some input configuration. More precisely:

**Definition 3.3.1** *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.*

Definition 3.3.1 implies that after *Precomputation* there is no more need to invoke the optimizer, because for any configuration input to the INUM the optimal plan can be found in the INUM Space. In order to make the *INUM Space* practical, we address the following problems:

1. Efficiently computing the INUM Space. For plan reuse to be effective, the size of the INUM Space must be much smaller than the number of configurations that must be examined. After verifying that the INUM Space actually has this property, we must provide an algorithm to compute the family of optimal plans specified by Definition 3.3.1.

2. Efficiently locating the optimal plan in the INUM Space for an input configuration.

The key intuition behind the INUM 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. different optimal for that query.

To see why the number of potentially optimal plans is small, consider a query accessing $n$ tables. The total number of plans for this query is $O(n!)$, but this does not imply that every plan has the potential to be optimal for some configuration. For example, plans that construct huge intermediate results will never be optimal. In addition, the optimality of the plan does not change very easily by changing index configurations, as it is strongly dependent on additional parameters, such as the intermediate result sizes. Thus an optimal plan for configuration $c_1$ might in fact remain optimal for a set of configurations that are "similar" to $c_1$. We show that the degree of plan reuse is such that the effort in precomputing the set of optimal plans is easily amortized by the huge number of optimizer calls that can be performed almost instantly afterward. Furthermore, we characterize the "families" of configurations that share each optimal plan and use this characterization to efficiently

identify the optimal plan that corresponds to a given input configuration.

In the remainder of this 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. After presenting our results for MHJ plans, Section 3.6 extends our approach to include NLJ plans.

Our two-step approach is necessary because there are differences in the way an index configuration affects the cost of NLJ and MHJ plans. We prove a linear formula to compute the cost of any MHJ plan given the index access costs. Linearity simplifies both the INUM construction and the selection of the optimal plan and we choose to present those results first. NLJ plans do not exhibit this linear dependence and accurate cost estimation requires more knowledge of the optimizer internal operation. We present a solution in Section 3.6.

## 3.4 Using Cached MHJ Plans

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

### 3.4.1 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 $C_{internal}$ 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:

$$C_p = C_{internal} + s_{T_1} + s_{T_2} + ... + s_{T_n} \tag{3.1}$$

Equation (3.1) assumes that $p$ uses *all* the indexes of $C$, otherwise the $s_{T_i}$ values are invalid. Notice that equation (3.1) does not imply that $p$ is the optimal plan for $C$. It just expresses the cost of any plan $p$ as a "function" of the input configuration $C$. Given a way to identify the optimal plan for $C$, equation 3.1 returns the correct query cost. Equation 3.1 can be very efficiently computed because $C_{internal}$ is already cached in the INUM Space and the $s_{T_i}$ parameters are computed on-the-fly by the optimizer.

The following conditions are necessary for the validity of equation (3.1).

1. $C_{internal}$ is independent of the $S_{T_i}$'s. If $C_{internal}$ 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, (3.1) is not a function of the $s_{T_i}$ variables.

3. Plan $p$ can actually use the indexes in $C$. If a plan expects a specific ordering (for instance, to use with a merge join) but $C$ does not contain an index to provide 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. 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 3.4.1** *A plan is* compatible *with a configuration and vice-versa if plan p can use the indexes in the configuration without any modifications to its internal structure.*

Assuming constraints (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.

### 3.4.2 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.

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. To account for this

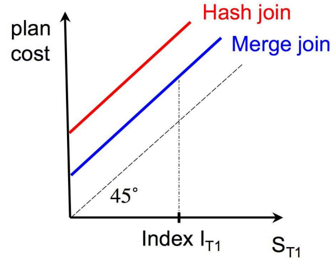


Figure 3.4: The cost functions for MHJ plans form parallel hyper-surfaces.

case, 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 3.3.1, 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 opti-

mizer 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 $C_{internal}$ 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 3.4.1** *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 $C$.*

**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 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.

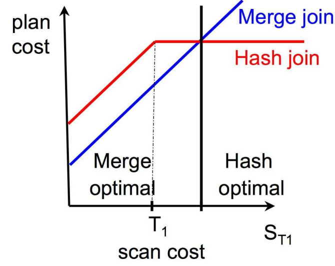


Figure 3.5: Modified plan comparison taking into account index I/O costs. The optimal plan for expensive indexes (to the right of the thick line) performs a sequential scan and uses a hash join.

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 3.4.1 by storing for each plan its region of optimality. The 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 3.4.1 the retrieved plan will be the optimal plan that accesses all the indexes in $C$. Like 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 3.4.1 for every subset of $C$ and return the plan with the lowest cost. Enumerating $C$'s subsets for $n$ tables requires in the worst case

$2^n$ iterations. Since each evaluation corresponds to a lookup based on a set of interesting orders and $n$ is fixed, the exponent does not hurt performance: For 5 tables, subset enumeration requires merely 32 arithmetic evaluations.

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 of the input configuration $C$, or equivalently use a specific subset $C'$. It can be shown that an index is not used only if it has an access cost that is too high. 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.

Parametric Query Optimization (PQO) techniques can be directly applied to piecewise linear cost functions like those in Figure 3.5, in order to directly find the optimal plan given the index access cost values. We omit the details of a PQO model due to lack of space.

## 3.5 Computing the INUM Space

Theorem 3.4.1 in Section 3.4.2 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 "empty" 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 3.4.1, 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 3.4.1, 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 INUM Space is $2^n$. For $n = 5$, 32 optimizer calls are sufficient for subsequently 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 [15]. Fortunately, there are ways to optimize the performance of INUM construction, so that it still outperforms optimizer-based ap-

proaches. 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 [15] 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 which 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 $C_{internal}$ 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 smaller tables is not as important.

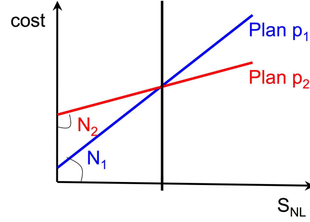


Figure 3.6: NLJ plan costs for a single table as a function of an index's $s_{NL}$ parameter (System R optimizer).

## 3.6 Extending the INUM

In this section we consider plans containing nested-loop join operators (NLJ plans) and explain why they require additional modeling effort. We present ways to incorporate NLJ plans in the INUM.

### 3.6.1 Modeling NLJ Plans

The cost of an NLJ plan can not be described by Equation (3.1) of Section 3.4.1. 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 [58]. For System R the cost of a plan using a 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 plans. 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 optimization tech-

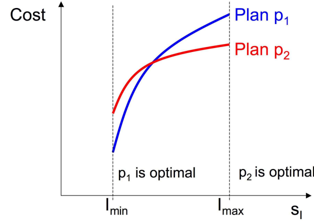


Figure 3.7: NLJ plan cost curves for a single table and an unknown cost function of a single index parameter $s_I$.

niques. In this dissertation 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.

### 3.6.2 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 3.6.1) that relates to its 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 plans 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 [1]. 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 3.4.1, 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 exist an $I_{min}$ and $I_{max}$ index per interesting order. The INUM performs an optimizer call for every $I_{min}$ and $I_{max}$ combination. This procedure results in more optimizer calls compared to the MHJ case, which required only a single call per interesting

[1]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 $I_{min}$.

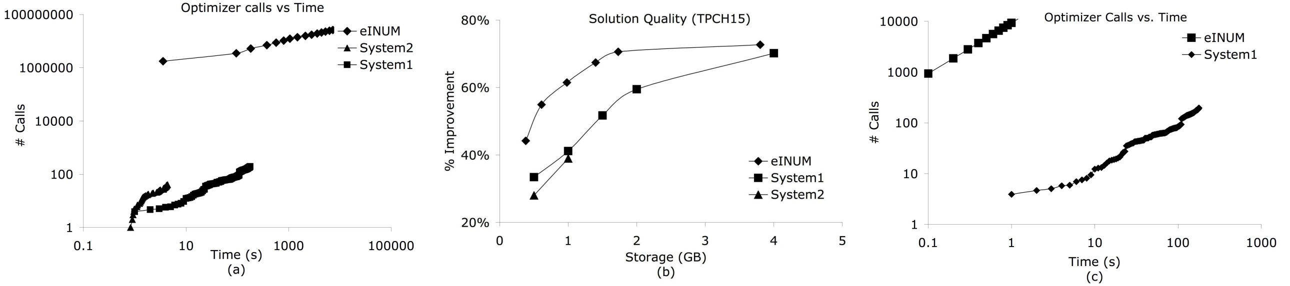


Figure 3.8: Experimental results for TPCH15. (a) Optimizer calls vs. time for an exhaustive candidate set (b) Recommendation quality (c) Optimizer calls vs. time for a heuristic candidate set

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, the 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 3.4.2). 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.

## 3.7 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 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 [15].

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 and the NREF protein database described in [19]. The NREF database consists of 6 tables and consumes is 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 based server with 4 gigabytes of RAM running Windows Server 2003 (64bit). We report both tuning running times and recommendation quality, that is computed using optimizer estimates. Improvements are computed by:
$\%improvement = 1 - cost_{indexed}/cost_{notindexed}$.

## 3.8 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.

### 3.8.1 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 optimization time: for each point in the horizontal axis, the graph shows the number of estimation calls up to that point. 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 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. 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 is 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* comparing with *System1*, in a logarithmic plot. *System1*, 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 precomputation* (Section 3.5), 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 [15]. Incorporating additional optimizations

would have reduced the precomputation overhead down, since it would allow a further reduction in the number of optimizer calls. even further.

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

#### INUM Precision

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 [15] or the benefit assignment method for the knapsack formulation [32] already trade accuracy for efficiency.

### 3.8.2 NREF Results

In this section, we present our results from applying *eINUM* with an exhaustive candidate index set on the NREF workload. NREF is dif-

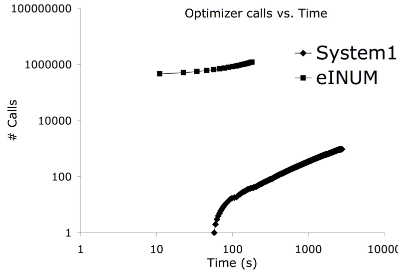


Figure 3.9: Optimizer calls vs.time for the NREF workload

ferent 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.

## 3.9 Conclusion

Index selection algorithms are built around the query optimizer, however the query optimization complexity limits their scalability. We introduce INUM, a framework that solves the problem of expensive op-

timizer 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 evaluate INUM in the context of a real commercial query optimizer and show that INUM improves enumeration performance by orders of magnitude. In addition, we demonstrate that being able to evaluate a larger number of candidate indexes through INUM improves recommendation quality.

# Chapter 4

# An Integer Linear Programming Approach to Automated Database Design

## 4.1 Introduction

Automated database design is a major challenge in building self-tuning database management systems. A major difficulty in the development of practical physical design algorithms is dealing with the huge number of design *features*, such as indexes or materialized views, that are relevant to an input workload and must be considered. A second difficulty is determining, given a (pruned) search space of candidate features, a combination that provides optimal performance for the workload that also satisfies resource constraints such as available storage. Theoretical studies in index and materialized view selection prove that determining optimal solutions is computationally hard [14].

An orthogonal problem is the design of a cost model in order to compare the merits of different design alternatives. Existing tools typically rely on the query optimizer for workload cost estimation under varying physical designs. The query optimizer provides an accurate cost model,

but each invocation is expensive in terms of execution time. Query optimizer performance directly affects physical design algorithms, as it increases their running times and limits their scalability in terms of the total number of alternative designs they can consider.

Existing techniques address the above challenges with carefully engineered tools that involve a multitude of application-specific heuristics. Modern physical design tools select promising candidates by separately analyzing each query in the workload [15, 2] and traverse the pruned search space in a cost-based, greedy fashion, during which additional candidates can be generated [9]. To avoid the overhead of the query optimizer, several approximation schemes are employed (such as local plan changes [9, 10]) that provide upper or lower bounds to the actual query cost values and are used to guide the heuristics.

Although heuristics have led to practical implementations, they have an unpredictable impact on solution quality. For example, the greedy search heuristic might be fast, but considering one (or a few) design features at a time ignores the *interaction* between features [73] and could easily be diverted to suboptimal solutions. By the same token, generating new candidates in parallel with the search [9] is prone to missing important candidates. The above inaccuracies are made worse by cost model approximations, that can easily mis-guide the search into suboptimal paths. There is currently no general analysis on the tradeoff between algorithm performance and solution quality, as the tradeoff is considered a necessary condition for practical tools.

This dissertation introduces a radically different approach to database physical design. Instead of immediately addressing the engineering of practical algorithms using heuristics we take a step back and model database physical design as a standard combinatorial optimization problem, that must in principle be optimally solved. Our model opens the

way for the application of a huge body of work in combinatorial optimization and operations research, that is successfully deployed for real-world, large-scale optimization problems in countless other domains, but not in databases.

We model physical design using an Integer Linear Programming (ILP) formulation, that captures the full complexity of the design problem, while admitting practical implementations. We couple our ILP abstraction with general, industry-strength optimization tools, such as CPLEX, that can find *optimal* solutions for very large problem instances occurring in practice. This chapter focuses on index selection, but the ILP formulation is applicable to other design features.

Our approach separates the optimization part of physical design, which is offloaded to ILP solvers, from the modeling part, that involves capturing critical model parameters (such as query costs) and requires database-specific knowledge. For the optimization part, we replace the greedy search heuristic used in previous approaches by generic, highly-efficient optimization engines that avoid the shortcomings of greedy search (such as local minima). State-of-the-art engines can handle very large problem instances consisting of thousands or tens of thousands of decision variables, or in our case, index combinations, in a very small amount of time. Contrary to heuristic search, we exploit both the performance of modern solvers and their optimality guarantees.

The problem now shifts to deriving an appropriate ILP model for a given physical design problem. In constructing an ILP model, we face the same difficulties as in developing a heuristic search algorithm. First, the ILP formulation cannot have unlimited size. Even for small problems, we can naively generate unreasonable ILPs, with millions or billions of variables. Second, even with practical problem sizes it is very expensive to solely rely on the query optimizer for cost estimation. We

therefore still need to adjust the model size and parameters to allow for efficient implementations.

What makes our approach different from previous work is that we show how to use the mathematical model to bound the impact of our heuristics to solution quality. This allows us to apply heuristics in order to improve running times in a *controlled fashion*, while providing a guarantee about the worst-case loss in solution quality. If the bounds obtained are not satisfactory, the precision of the ILP formulation can be increased in order to improve the quality of the solution (with a necessary increase in running time). Previous approaches, based on heuristic search, can not provide such guarantees simply because it is not possible to provide optimality bounds for the greedy traversal.

In this chapter, we make the following contributions:

1. We show how to take advantage of existing Integer Linear Programming (ILP) solvers for database design. The efficiency of state-of-the-art optimization engines allows us to handle very large design problems with realistic databases and workloads and find *optimal* solutions within seconds.

2. We couple the increase performance of ILP solvers with a fast query cost estimation module, based on our work on optimizer plan reuse. Our cost estimation module, the Index Usage Model (INUM) (Chapter 3), provides query cost estimation results with three orders of magnitude better performance compared to the query optimizer, but the same precision.

3. We introduce a technique for eliminating indexes and index combinations from consideration, that considerably reduces problem sizes. Contrary to previous techniques, we can provide *per-instance* optimality guarantees, *before* solving the problem. In this way

database administrators can balance solution quality with solution efficiency.

4. We introduce a technique for minimizing the number of optimizer calls required, through query cost approximation. Our technique performs even fewer calls compared to the INUM and allows the computation of approximate solutions fairly quickly, with guaranteed optimality bounds. If the provided bounds are not satisfactory, further optimization calls can be performed. Using this technique, we were able to process workloads consisting of 1000 queries, performing only 2 optimizer calls per query, while provably staying within 10% of the optimal. Due to our approximation, we achieved a 10x speedup and 10x improvement in quality compared to existing tools.

This chapter is organized as follows. Section 4.2 formalizes index selection using our ILP formulation. Section 4.3 describes the index selection tool architecture we use to exploit our ILP formulation. Section 4.4 details the cost approximation schemes we use to improve performance *with a predictable impact on quality.* Sections 4.5 and 4.6 detail our experimental evaluation, while Section 5.7 presents concluding remarks.

## 4.2 An ILP Model for Index Selection

In this section we introduce an integer linear programming formulation that captures the full complexity of the index selection problem.

### 4.2.1 Mathematical Formulation

Consider a workload consisting of $m$ queries and a set of $n$ indexes $I_1$-$I_n$, with sizes $s_1$-$s_n$. We want our model to account for the fact that a query has different costs depending on the *combination* of indexes it uses. A *configuration* is a subset $C_k = \{I_{k1}, I_{k2}, ...\}$ of indexes with the property that all of the indexes in $C_k$ are used by some query.

Let $P$ be the set of all the configurations that can be constructed using the indexes in $I$ and that can potentially be useful for a query. For example, if a query accesses tables $T_1$, $T_2$ and $T_3$ then $P$ contains all the elements in the set (indexes in $I$ on $T_1$) $\times$ (indexes in $I$ on $T_2$) $\times$ (indexes in $I$ on $T_3$).

The cost of a query $i$ when accessing a configuration $C_k$ is $c(i, C_k)$ and $c(i, \{\})$ denotes the cost of the query on an unindexed database. We define the *benefit* of a configuration $C_k$ for query $i$ by $b_{ik} = max(0, c(i, \{\}) - c(i, C_k))$.

Let $y_j$ be a binary decision variable that is 1 if the index is actually implemented and 0 otherwise. In addition, let $x_{ik}$ be a binary decision variable that is equal to 1 if query $i$ uses configuration $C_k$ and 0 otherwise.

Using $x_{ik}$ and $b_{ik}$, the benefit for the workload $Z$ is

$$Z = \sum_{i=1}^{m} \sum_{k=1}^{p} b_{ik} \times x_{ik} \tag{4.1}$$

where $p = |P|$. The values of $x_{ik}$ depend on the values for $y_j$: We cannot have a query using $C_k$ if a member of $C_k$ is not implemented. Also, we require that a query uses at most one configuration at a time. For instance, a query cannot be simultaneously using both $C_1 = \{I_1, I_2, I_3\}$ and $C_2 = \{I_1, I_2\}$. Finally, we require that the set of selected indexes consumes no more than $S$ units of storage. Thus the formal

| | No index | Single index | | Index pairs |
|---|---|---|---|---|
| $Q_1$ | cost=120 | $C_1$={$I_1$} $S_1$=100 | $C_2$={$I_2$} $S_2$=100 | $C_3$={$I_1$,$I_2$} |
| | | $c_{11}$ = 90 $b_{11}$= 30 | $c_{12}$ = 90 $b_{12}$= 30 | $c_{13}$= 45 $b_{i3}$ = 75 |
| $Q_2$ | cost=120 | $C_4$={$I_3$} $S_3$=100 | $C_5$={$I_4$} $S_4$=100 | $C_6$={$I_3$,$I_4$} |
| | | $c_{24}$ =85 $b_{24}$= 35 | $c_{25}$ =85 $b_{25}$= 35 | $c_{26}$ = 47 $b_{26}$= 73 |

Figure 4.1: Index selection example.

specification of the index selection problem is as follows.

$$maximize \quad Z = \sum_{i=1}^{m} \sum_{k=1}^{p} b_{ik} \times x_{ik} \tag{4.2}$$

subject to

$$\sum_{k=1}^{p} x_{ik} \leq 1 \quad \forall i \tag{4.3}$$

$$x_{ik} \leq y_j \quad \forall i, \ \forall j, k : I_j \in C_k. \tag{4.4}$$

$$\sum_{j=1}^{n} s_j \times y_j \leq S \tag{4.5}$$

Constraints (4.3) guarantee that a query uses at most one configuration. Constraints (4.4) ensure that we cannot use a configuration $k$ unless all the indexes in it are built and constraint (4.5) expresses the available storage.

Figure 4.1 shows an example with 2 queries and 4 indexes, listing all the relevant configurations for each query. Assume only indexes $I_1$ and $I_2$ are relevant to $Q_1$, whose cost varies depending on whether uses a single-index configuration ( $c_1$ or $c_2$) or a pair ($c_3$). The same holds for $Q_2$ and indexes $I_3$ and $I_4$.

Assume we want to optimize workload benefit given total storage capacity of $S$=200 units.

$$minimize\ Z = b_{11} \times x_{11} + b_{12} \times x_{12} + b_{13} \times x_{13} + \\ + b_{24} \times x_{24} + b_{25} \times x_{25} + b_{26} \times x_{26} \qquad (4.6)$$

subject to

$$\sum_{k=1}^{3} x_{1k} \leq 1, \quad \sum_{k=4}^{6} x_{2k} \leq 1,$$
$$x_{11} \leq y_1,\ x_{12} \leq y_2,\ x_{13} \leq y_1,\ x_{13} \leq y_2,$$
$$x_{23} \leq y_3,\ x_{24} \leq y_4,\ x_{35} \leq y_3,\ x_{36} \leq y_4,$$
$$\sum_{j=1}^{4} s_j \times y_j \leq 200$$

By inspection we determine the optimal solution

$$y_1 = 1,\ y_2 = 1,\ y_3 = 0,\ y_4 = 0,$$
$$x_{11} = 0,\ x_{12} = 0,\ x_{13} = 1,\ x_{24} = 0,$$
$$x_{25} = 0,\ x_{26} = 0$$

The set of indexes $I_1$ and $I_2$ is preferable because their combination has a large benefit for $Q_1$ and outperforms any other alternative. Notice that the commonly used greedy search would fail to identify t he optimal solution. In the first iteration it would pick index $I_3$ and in the second $I_4$.

The exact solution provided by the ILP formulation is optimal *for the given initial selection of indexes.* If we were to include all the possible indexes that are relevant to the given workload, it would give us the globally optimal solution.

Considering the set of all the possible indexes is prohibitively expensive and thus a candidate selection module is necessary. The ILP approach is flexible in that we can use it with an arbitrary candidate index set.

### 4.2.2 Supporting Updates & Clustered Indexes

The ILP formulation of Section 4.2 can be extended to handle updates in the workload (SQL *INSERT*, *UPDATE* or *DELETE* statements). We model an update statement as a sequence of two sub-statements, "select" and '"modify". The "select" part is just another query selecting the set of rows to be modified or deleted and thus is handled by the formulation of Section 4.2 (*INSERT* statements do not have a selection part and thus get a zero benefit value for all configurations). The "update" part is a statement that simply updates the set of rows returned by the "select" part. It has a different behavior, because an index configuration $C_k$ can have a *negative benefit* for the update part, because of the additional cost for updating the relevant indexes in $C_k$. Specifically, the benefit $b^U_{lk}$ of configuration $C_k$ for the update sub-statement $U_l$ is

$$b^U_{lk} = cost_{update}(l, \{\}) - cost_{update}(l, C_k) \tag{4.7}$$

$cost_{update}(l, )$ is zero, while $cost_{update}(l, C_k)$ is equal to the sum of the costs for updating all the indexes in $C_k$.

$$b^U_{lk} = - \sum_{I_j \in C_k} cost_{update}(l, I_j) \tag{4.8}$$

Generally, for every index $I_j$ we can associate a (negative) benefit value $-f_j$ which is computed by summing up all the $-cost_{update}(l, I_j)$ values over all the update statements $U_l$ and corresponds to the update overhead introduced by that index. To model updates, we only need

to modify the objective function of Section 4.2 (Equation 4.2) to take into account the negative benefit values $f_j$.

$$maximize \;\; Z = \sum_{i=1}^{m+m_1} \sum_{k=1}^{p} b_{ik} \times x_{ik} - \sum_{j=1}^{n} f_j \times y_j \tag{4.9}$$

Equation 4.9 describes the workload benefit in the presence of $m$ queries and $m_1$ update statements. The second term simply states that if index $I_j$ is constructed as part of the solution, it will cost $f_j$ units of benefit to maintain it in the presence of the $m_1$ update statements.

Supporting clustered indexes is straightforward with our model. A candidate clustered index is yet another index in the candidate set, one that contains all the attributes in a relation. We allocate a $y_j$ variable to it as usual. It also participates in combinations naturally. The size of clustered indexes is artificially set to 0 (as no additional space is required to sort a table). For each table $T$ we restrict the set of clustered indexes on it, say $\{y_{T_1^c}, y_{T_2^c}, ...y_{T_l^c}\}$ so that only one clustered index is picked:

$$y_{T_1^c} + y_{T_2^c} + ... + y_{T_l^c} \leq 1 \tag{4.10}$$

## 4.3 An ILP-based Index Selection Tool

In the previous sections we presented an ILP formulation that completely describes the index selection problem. In this section we discuss the architecture of a practical ILP-based index selection tool.

Figure 4.2 details the components that are used in our tool. All the modules except for the ILP solver are used in the construction of the model, deciding the $x_k$ and $y_j$ variables and computing the benefit values $b_k$, all described in Section 4.2. Once the model is constructed, the ILP solver is used to determine the optimal solution.

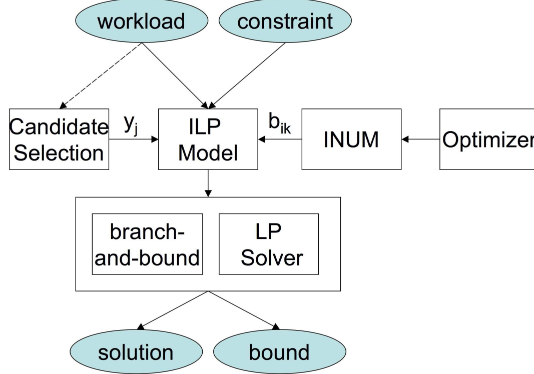


Figure 4.2: Architecture for an ILP-based index selection algorithm.

The *Candidate Selection* and *Combination Selection* modules allow the determination of the $x_k$ and $y_j$ decision variables from the problem at hand. For each combination $x_k$ participating in the model, the *Cost Estimation* module determines the query costs and corresponding $c_k$ benefit values. *Cost Estimation* is typically based on the query optimizer. In our system, we couple the query optimizer to the *Index Usage Model (INUM)*, a mechanism we have developed for improving the performance of query cost estimation through caching and reusing of optimizer computation. The INUM is three orders of magnitude faster than a query optimizer call, while providing exactly the same result. Section 4.3.2 provides an overview of the INUM, while a detailed description appears in Chapter 3. The completed ILP representation is consequently input to the ILP solver.

The overall performance depends on the numbers of $y_j$ and $x_k$ variables, which control the problem size and consequently the optimization time and the time spent in the cost estimation module. Our ILP formulation is advantageous compared to existing approaches in that the efficiency of modern ILP Solvers and of the INUM allows us to consider very large numbers of decision variables (candidate indexes and index combinations). In our experiments, we have been able to solve ILP instances considering up to 110,000 candidate indexes and 3.2 million

combinations within minutes.

Furthermore, our ILP formulation allows for a particularly attractive modular design, where the impact of optimizations in each module on the final solution can be analyzed and quantified. Each module can apply cost-based pruning (for *Candidate and Combination Selection*) and approximations (for *Cost Estimation*), to improve performance.

In the remaining subsections we describe the components of Figure 4.2 in more detail. We start with the ILP Solver and the INUM, which are the foundations of our technique, providing efficiency and scalability and continue with the *Candidate Selection* and *Combination Selection* modules, that ensure that the resulting ILP formulation accurately reflects the index selection problem at hand.

### 4.3.1 ILP Solver

The *ILP Solver* module of Figure 4.2 takes as input the ready ILP formulation corresponding to an index selection problem and computes the optimal solution. While ILPs are NP-hard in the worst case, modern ILP solvers can efficiently optimize very large problem instances. The index selection problem in databases is particularly amenable to the optimization algorithms in modern optimization engines. Using CPLEX, an industry-strength optimization tool, allowed us to solve index selection ILPs with millions of decision variables always in less than a minute.

For the cases where a particular problem instance proves to be more "difficult and optimization does not finish within a given time threshold, the solver can be interrupted at any time. The partial solution reached by the solver is in this case suboptimal, however there is an immediate way to estimate a bound from the optimal solution. Solving the Linear Programming (LP) relaxation of an ILP problem [12] is

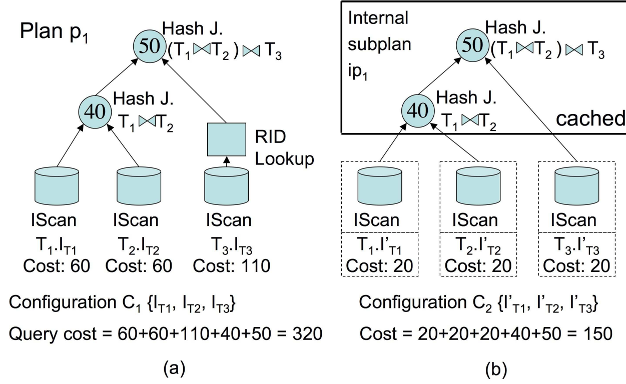


Figure 4.3: Illustration of plan reuse with INUM. (a) The optimal plan $p_1$ for configuration $C_1$. (b) The cost for $C_2$ is computed by reusing the cached internal nodes of plan $p_1$ and adding the costs of the new index access operators. This example is valid under the constraints described in Section 4.3.2.

very efficient (there exist polynomial algorithms for LP problems) and the LP solution provides a bound for the objective function of the ILP problem. The computed LP bounds are tight for index selection problems. For our experiments, those instances taking more than 1 minute to optimize were interrupted, with a bound that was always less than 0.2%.

### 4.3.2 The Index Usage Model

The Index Usage Model is a framework for efficient and accurate query cost estimation. Given a query and an index combination, it computes a value for the query cost that is the same as the one that would have been returned by an optimizer call with the same input. The INUM computes cost estimates very efficiently, by not invoking the query optimizer, except for a small number of "seed" calls. The query plans returned from the "seed" calls are used by the INUM to accurately compute query costs for all the other calls, without additional optimizer invocations.

Since the optimizer is not involved for the large majority of cost

estimations, the INUM can achieve 3 orders of magnitude faster performance while maintaining optimizer precision. The INUM is independent of the index selection algorithm used, however it is a direct match for our ILP-based approach, that relies on considering large numbers of index combinations to achieve good solution quality.

The intuition behind the INUM is that although the number of possible index combinations ($x_k$ variables in our model) relevant to a given query can be very large (thousands or tens of thousands), the number of *different optimal query execution plans* and thus the range of different optimizer outputs is much smaller (Consider for example the number of different ways to execute an SPJ query on two tables). Therefore it is more cost effective to *reuse* the set of optimal plans, than take the cost of computing the same plan multiple times.

Figure 4.3 shows an example of plan reuse in INUM. In Figure 4.3 (a), the optimizer is used to obtain the query execution plan for query $Q$, using the indexes in combination $C_1$. The INUM caches the internal structure of the plan, the *Internal Subplan* $ip_1$ shown in Figure 4.3 (b), in order to reuse it for other input combinations. Assume that an index selection tool asks for the cost of an index combination $C_2$, whose indexes do not contain the join columns (and thus can not be used by the optimizer to facilitate merge or nested loop joins). The internal structure of plan $p_1$ is applicable to $C_2$, since it contains only hash joins and it can be shown that it is in fact the plan that the optimizer would have selected for $C_2$: If the optimizer was to return a different plan, say $p_2$, then $p_2$ could have been used for $C_1$ as well, but we know this is not possible since $p_1$ is optimal. Therefore, the cost of $Q$ under $C_2$ can be computed by reusing the *cached* cost of $p_1$ and adding the individual access costs for the indexes in $C_2$. The index access cost can be efficiently computed using the relevant optimizer cost models without invoking the bulk of the query optimization process.

In the example of Figure 4.3, plan $p_1$ can be reused with input combinations that do not involve the join columns of $Q$ and thus save the overhead of multiple optimizer calls that we know would have returned plan $p_1$. If the input combination involves the join columns however, reusing $p_1$ might result in erroneous estimation, as the optimizer is likely to select more efficient join algorithms and join orders. The INUM accounts for such cases by performing additional optimizer calls to retrieve the set of *all possible* optimal plans. We developed the *matching logic* that allows INUM to distinguish whether a cached plan is appropriate for reuse with a given input combination, or a new optimizer call must be performed and cached. INUM's matching logic is based on simple observations about optimizer operation and can be implemented without additional interfaces. Chapter 3 of this dissertation presents a detailed description of INUM, along with an evaluation of its correctness and performance advantages using a real commercial optimizer.

We use INUM to accurately compute query costs and the corresponding $c_k$ variables for thousands of combinations, while achieving lower total running times compared to existing tools. We report both the time consumed in obtaining INUM's cost estimates and the time spent by INUM in obtaining and caching optimal plans from the query optimizer.

### 4.3.3 Candidate Selection

If it were possible to allocate a $y_j$ variable for every possible index that is relevant to a given workload, the formulation of Section 4.2 would specify the full index selection problem and allow us to identify the optimal solution[1]. Since every subset of attributes for every table

[1] Assuming we could also exhaustively generate all possible combinations ($x_{ik}$ variables as well.

referenced by the workload is a potential index, exhaustively computing all possible relevant indexes in a workload is impractical. The goal of *Candidate Selection* is to determine a set of promising *candidate* indexes, that is likely to contain a superset of the optimal solution, although there are no guarantees.

We built a *Candidate Selection* module using a strategy that generates thousands or tens of thousands or candidates, a number at least an order of magnitude higher compared to existing index selection tools. Our system is designed to provide high performance for such large candidate set sizes and thus has the advantage, over existing work, that it minimizes the chances of "missing" some important index. Furthermore, we note that our system can incorporate any previously proposed candidate selection technique, such as the "per-query-optimal" indexes of [15], or cost-based index transformations, adapting techniques from [9].

#### Per-query-relevant Index Selection

We model an index on a table $T$ as a tuple $(K, S)$ (using the same notation as in [10]), where $K$ and $S$ are subsets of $T$'s attributes. $K$ consists of the *key* attributes and thus is order-sensitive, while $S$ is a suffix, insensitive to attribute ordering.

Our strategy, called *per-query-relevant* candidate selection, operates on a per-query basis. For each query $Q$, referencing tables $T_1, ...T_N$, we build attribute subsets $S_{T_1}, ..., S_{T_N}$, where $S_{T_i}$ consists of all the attributes in $T_i$ referenced by $Q$. We then proceed to build an index for each subset of attributes in each $S_{T_i}$, also taking into account the different orders in the key part of the index. The number of combinations that can be generated from a set $S_{T_i}$ depends on its size and also on the number of attributes in the key. Assuming a key length of 1, gives

us $|S_{T_i}| \times 2^{|S_{T_i}|}$ possible indexes.

The above technique is likely to also build indexes that although syntactically relevant, are useless performance-wise. Consider for example indexes on a small subset of $Q$'s select-clause attributes. Such indexes do not take advantage of where-clause predicates and potentially incur a high RID lookup overhead, since they require additional accesses to the base tables. We use a *cost-based filtering* step to eliminate such expensive indexes. We compute the cost of *scanning* each candidate with respect to $Q$ and eliminate those that have a scan cost greater than or equal to $a \times cost(Q, \{\})$, where $0 \leq a \leq 1$. For $a = 1$, we eliminate indexes with a scan cost larger than the total cost of $Q$ when running unindexed, as these indexes are guaranteed to not be used [2]. By reducing the value of $a$, for instance setting $a = 0.8$, we improve the effectiveness of the filter with a low probability of missing important indexes: For $a = 0.8\%$, a "pruned" index can not be useful unless the cost of executing the rest of the query is "compressed" to less than 20% of its original value. Even in this case, the benefit lost by ignoring such and index would be small.

#### Effect of Candidate Selection on Solution Quality

Candidate selection plays an important role in the ILP formulation, as the selection of the $y_j$ variables affects solution optimality. The 'globally optimal solution is obtained only by exhaustively generating the set of all possible indexes. By pruning even a single candidate, we generate a "restricted" ILP instance that is different from the exhaustive one. By solving the "restricted" ILP, even in an optimal fashion, the resulting solution might be different from the "globally" optimal one. Thus, it is important to note that the solution specified by an ILP formulation

[2]Notice that the scan cost includes possible predicates in $Q$ that might take advantage of the index.

is essentially the optimal solution *with respect to the provided set of candidate indexes.*

Computing optimality bounds for candidate selection algorithms is inherently more difficult, compared to analyzing the other approximations described in this chapter. In order to estimate a benefit value for a single index $y_j$ any approach would need at least one optimizer (or INUM) call (probably more if we consider all possible combinations $x_{ik}$ using this index). The number of possible indexes is at least exponential to the number of attributes in a table and thus grows very large very quickly: for instance, a 20 attribute table generates about 20 million candidate indexes.

Due to its exponential size, enumerating an exhaustive set of candidates might be impractical from a performance perspective. Pruning in this case can not be simply cost based, as we will have to eliminate sets of indexes *even without considering their costs.* While such an approach is possible for *Combination Selection* presented in the following section, in this case it is difficult to characterize the relationship between index properties and query performance and derive bounds on the latter.

We chose to deal with this lack of mathematical structure by simply generating as many candidates as possible, thus reducing the probability of missing an index that could be important. The only restriction that we impose is that all the attributes in a candidate index are referenced in at least one query of the workload (thus excluding indexes that have some attributes referenced by one query and some by another). Our techniques capture all the possible "covering" indexes for a query and their subset, thus it is guaranteed to contain the optimal indexes for a query and is in fact a superset of the per-query-optimal approach used in the literature. We do not take into account "merged" indexes, however our technique can easily be adapted to accommodate them.

Our experiments with the TPC-H workload suggested that, starting with our per-query-relevant index selection, merging afterwards does not significantly improve solution quality.

### 4.3.4 Combination Selection

Constructing the ILP model requires "assembling" index combinations (variables $x_{ik}$) from the candidate indexes (variables $y_j$) selected by the *Candidate Selection* module. Exhaustively iterating over all possible combinations is likely to generate a huge number of variables. Besides increasing the problem size and optimization times, generating a large number of variables also requires more work to estimate all the $b_{ik}$ variables. The combination selection module is responsible for reducing the number of $x_{ik}$ variables through pruning.

In this section we describe two combination selection approaches that improve the efficiency of ILP-based index selection and show how to bound the quality loss resulting from pruning. *Cost-based pruning* intuitively eliminates all combinations that are unlikely to participate in the optimal solution because they offer very low benefits. *Pruning based on table-subsets* keeps only those combinations that involve indexes on certain 'interesting subsets of tables, motivated by the observations that only a few of the tables are responsible for most of the query costs. In both cases, the ILP formulation allows us to estimate the worst-case benefit loss by combination selection.

#### Cost-based Pruning

In the ILP formulation, omitting a combination $x_{pruned}$ from consideration means a-priori determining the value of $x_{pruned}$:

$$x_{pruned} = 0 \tag{4.11}$$

The resulting "simplified" ILP model has one less variable and thus is easier to solve. However if in the optimal solution $x_{pruned} = 1$ then our pruning was wrong, since we interfered with the optimal solution to the original problem.

Cost-based pruning iterates over all possible $x_{ik}$ variables for a problem and eliminates those that offer a relative improvement of less than $e\%$, where $e$ is a user-defined parameter. For example, if $e = 20\%$, for every query $Q$, every relevant combination $x_k$ with a benefit value $c_k \geq 0.2 \times cost(Q, \{\})$ will be included in the model and everything else will be omitted. We can in fact specify multiple threshold values $e_i$, one per query, and apply each threshold to the range of $x_k$ variables corresponding to each query. Multiple threshold values allow us to prune more combinations from the cheaper queries and intuitively, have less impact to the overall achievable improvement.

How does pruning affect the quality of the derived solution? Let $Z*$ denote the optimal value for the simplified objective function (after pruning) and $Z$ be the optimal objective function value for the original problem. Naturally, $Z* \leq Z$ and we can prove that

$$Z - Z* \leq \sum_{i=1}^{m} e_i \times cost(Q_i, \{\}) \tag{4.12}$$

Equation 4.12 states that the loss in quality is bounded by the unindexed query costs weighted by the individual thresholds and suggests the adaptive strategy of keeping low thresholds for relatively expensive queries, while aggressively pruning cheaper queries.

To prove Equation 4.12 we first show that omitting a single combination $x_{k_{pruned}}$ results in a reduction in the objective value function of at most $c_{pruned}$. Let the original problem have a solution vector $\mathbf{X} = (x_0, x_1, ...x_K, y_0, y_1, y_N)$ and an objective function value $Z$. Also, let the simplified problem have a solution vector $\mathbf{X}^* = (x_0^*, x_1^*, ..., x_K^*, y_0^*, ..., y_N^*)$

and an objective value function $Z^*$. Our goal is to show that

$$Z - Z^* \leq c_{k_{pruned}} \tag{4.13}$$

If $x_{k_{pruned}} = 0$, then the pruned combination is not used in the optimal solution and thus $Z - Z^* = 0$ and the theorem holds. If $x_{k_{pruned}} = 1$, then the solution to the simplified problem is suboptimal with respect to the original. If the two solutions agree in any other value, thus $\forall k \neq k_{pruned} : x_k = x_k^*$ then $Z - Z^* \leq c_{k_{pruned}}$ and the theorem holds.

How is it possible that the two solutions do not agree in all of their variables? It could be the case, for example, that omitting $x_{k_{pruned}}$ results in the corresponding indexes becoming less "attractive" and being omitted as well. Now the space released by the omitted indexes is allocated in a different way, resulting in an entirely different solution for the simplified problem. Even if this is the case, we can still prove (4.13. Consider a solution vector $\mathbf{X}^+$, derived from $\mathbf{X}$ by setting $x_{k_{pruned}} = 0$ and let $Z^+$ be the objective function value for $\mathbf{X}^+$ and leaving all other variables the same. The objective function values for the three solutions are related as follows:

$$Z - Z^+ \leq c_{k_{pruned}}. \tag{4.14}$$

$$Z^+ \leq Z^* \tag{4.15}$$

Equation (4.15) holds, otherwise $Z^*$ would not have been optimal. Equation (4.13) follows from (4.14), (4.15).

With equation 4.13,it is easy to see that if for query $Q_i$ we prune all the combinations with benefit values $c_k \leq e_i \times cost(Q_i, \{\})$, the maximum loss in benefit is $e_i \times cost(Q_i, \{\})$. Note that the benefit loss for a single query is not additive, as the query can only use a single combination. Repeating the pruning process for each query however is additive, hence equation 4.12.

**Table-Subsets**

*Cost-based pruning* reduces the problem size, but the costs of all combinations must still be computed through the *Cost Estimation* module. We propose a pruning technique based on *table-subsets*, where we keep index combinations on the most expensive tables. We are based on the observations that not tables are equally costly to access: In fact, table sizes and row counts are highly non-uniform, especially in decision support databases and star-schemas such as TPC-H. In a typical TPC-H statement, most of the benefit is obtained by indexing the largest tables, *LINEITEM* and *ORDERS*, as they are the most expensive to access. Thus *LINEITEM* and *ORDERS* form an "interesting" table subset, on which index combinations are likely to yield large benefit values. Indexes on the smaller tables, such as *NATION*, *REGION* and *SUPPLIER* might have some benefit, however this benefit will be smaller. The quality loss from not including indexes on such tables in combinations would be negligible.

We next quantify the previous statement. Consider a query $Q$ and let $T$ be a table subset that we wish to keep in the ILP formulation and $P$ be a set of tables that can be "pruned". This means that for $Q$ we can consider any combination of indexes built on tables in $T$ but no combinations involving tables in $P$. Notice that we can define a separate table subset for each query, consisting, for example, of the one or two of the most expensive tables referenced by $Q$. Naturally, for cheap queries we can consider smaller subsets (since anyways the benefit loss will be small), while for most expensive queries we want to preserve more information by looking at largest subsets.

Consider first the set of combinations $X_P$ that contains indexes only on tables in $P$. Every combination $x_k \in X_P$ will get pruned. Following the results of the previous section, the benefit loss for $Q$ will be bounded

by

$$\max_{x_k \in X_P} b_{Q,x_k} \tag{4.16}$$

If we properly select the table subset $T$ for $Q$, then the tables in $P$ will not be so important thus the maximum benefit loss will be small and thus can be ignored. As an example, consider the maximum benefit obtainable through index combinations on the set $P = \{NATION, REGION\}$. While the effect of indexes on $NATION$ or $REGION$ is small, considering one index on each table increases the number of possible combinations by 4.

Besides combinations exclusively on $P$, there exists a set of combinations $X_{PT}$, that contain some indexes on tables in $T$ and some indexes on tables in $P$. This set also gets pruned. Based on the reasoning followed above for the $P$, we might erroneously reach the conclusion that the benefit loss will be

$$\max_{x_k \in X_P T} b_{Q,x_k}. \tag{4.17}$$

Since, $X_{PT}$ contains combinations with indexes also on "important" tables, so this bound could be quite large.

Fortunately, although the set $X_{PT}$ gets pruned, there also exists the set $X_T$, consisting of combinations with indexes exclusively on tables in $T$, that gets preserved. Every combination is $X_{PT}$ that gets pruned has a smaller "subset" combination in $X_T$ that gets preserved, the only difference being that the latter lacks the indexes on the "unimportant" tables. The implication is that most of the benefit of combinations in $X_{PT}$ is preserved by the combinations in $X_T$.

This intuition is formalized as follows. Let $X_{opt}$ be the optimal solution obtained for the problem without any pruning and $Z_{opt}$ the corresponding objective function. Let $x_{pruned}$ be a decision variable corre-

sponding to a combination in $X_{PT}$ and $x_{pruned} = 1$ in $X_{opt}$. By pruning every combination in $X_{PT}$ we in fact set $x_{pruned} = 0$ which leads to a suboptimal solution. Let $X^*_{opt}$ be the resulting sub-optimal solution and $Z^*_{opt}$ be the corresponding objective function function. Obviously $Z^*_{opt} \leq Z_{opt}$ and our goal is to estimate $Z_{opt} - Z^*_{opt}$. Now consider another solution $X^+$, that is constructed from $X_{opt}$, by setting $x_{pruned} = 0$ and setting $x_{kept} = 1$, where $x_{kept}$ is a combination in $X_T$ that is a "subset" of the pruned one, without any index on the "unimportant" tables. Notice that if $X_{opt}$ is feasible then also $X^+$ is feasible (as it contains a subset of $X_{opt}$'s indexes) and let $Z^+$ be its benefit. We now have: $Z^+ \leq Z^*_{opt} \leq Z_{opt}$. We can now bound the quality loss as follows:

$$Z_{opt} - Z^*_{opt} \leq Z_{opt} - Z^+ \tag{4.18}$$

Since $X_{opt}$ and $X^+$ differ only in the $x_{pruned}$ and $x_{kept}$ values, Equation 4.18 becomes:

$$Z_{opt} - Z^*_{opt} \leq Z_{opt} - Z^+ \leq b_{pruned} - b_{kept} \tag{4.19}$$

Any of the combinations in $X_{PT}$ could potentially be in the optimal solution, so the worst-case loss for a single query becomes the maximum benefit difference between a pruned and its corresponding kept combination. Using the index $k$ for the various combinations in $X_{PT}$ and the corresponding subset combinations in $X_T$ we can rewrite Equation 4.19 as:

$$Z_{opt} - Z^*_{opt} \leq Z_{opt} - Z^+ \leq \max_{x^k_{pruned} \in X_{PT}} b^k_{pruned} - b^k_{kept} \tag{4.20}$$

The loss for multiple queries is additive, however notice that the definition of the $P$ and $T$ table-subsets for each query might be different. What equation 4.20 says is that with the ILP formulation, the drop in

solution quality is mathematically bounded by how close we approximate $b_{prune}$ by $b_{kept}$ or, at a higher level, by the relative "importance" of the tables in $P$ and $T$ and the characteristics of individual queries. Notice that although this intuition has been used in previous techniques, for example to simplify cost estimation [15], the ILP framework allows us to guarantee a maximum quality loss from its usage. Existing algorithms, such as greedy search, cannot provide such guarantees: greedy search can be arbitrarily bad and certainly approximating costs by ignoring certain tables does not improve things. In addition, the ILP formulation allows us to "customize" the approximation in a per-query fashion: More combinations can be allocated to queries that are most expensive or will benefit the most, thus optimizing the overall algorithm performance.

Another implication of the above argument is that, if the performance of a given table subset assignment is satisfactory, we can always "correct" by adding more tables in the $T$ set and thus add more combinations for queries. In the case of the ILP formulation this "resolution increase" is guaranteed to improve the solution (if a better solution exists) and this increase will be equal to the increase in the benefits caused by the addition of the extra indexes. On the other hand, existing heuristics and greedy search are again not guaranteed to reach a better solution, although the chances might be better.

Finally, a related question is: Can we precompute the bounds of Equation 4.20 before solving, so that we can "guide" the table-set heuristic? Computing the bound requires computing all possible $b^k_{pruned}$ values, which defeats the whole purpose of table-based pruning! Alternatively, we could approximate the bound of 4.20 by computing the benefit of the *optimal* combination for $Q$, $b_{optimal}$. The optimal combination will most likely belong to $X_{PT}$, as it will involve indexes from all the tables accessed by $Q$. For a given table-subset selection, we can

pick the subset combination in $X_T$ that has the lowest benefit $b_{kept}^{min}$. The bound then becomes:

$$Z_{opt} - Z_{opt}^* \leq Z_{opt} - Z^+ \leq b_m ax - b_{kept}^{min} \tag{4.21}$$

This bound is relatively pessimistic, however it is still indicative of the quality of the approximation and it allows us to fine-tune the selection of $P$ and $T$ depending on the query costs.

## 4.4 Approximate Cost Estimation

The *Cost Estimation* module computes for each combination $x_{ik}$ a benefit value $b_{ik}$. There are several approaches to cost estimation. The query optimizer provides the most accurate query estimates and is the cost model used in current systems [15, 2, 1]. Unfortunately query optimization is a time-consuming process and is very expensive to use for large number of combinations.

Since for database design it is desirable to evaluate as many combinations as possible, previous work introduces approximations, such as the *local transformation* approach of [9, 10] that avoids accessing the optimizer in most cases and thus is more efficient, but does not provide accurate results. Specifically, the local transformation approach returns *lower bounds* for the benefit of a particular combination.

Finally, we develop the Index Usage Model (INUM), a cost estimation technique that works by caching and reusing a small number of key optimizer invocations (See Section 4.3.2 for an overview and Chapter 3 for a detail description) to efficiently provide accurate cost estimates. Although primarily designed for accurate cost estimation, we can considerably reduce the INUM setup time by sacrifice some accuracy. The INUM can also be used to provide lower bounds instead of accurate benefits, by controlling the number of plans cached in the INUM Space. In

the worst case, we can cache only a single plan per query and use it to produce cost estimates (requiring a single optimizer call per query). If we wish to improve accuracy, we can alternatively cache and reuse more plans, although this might require more optimizer calls. To explore the performance-quality tradeoff exposed by the INUM, we need to answer the question: *how much accuracy do we need for index selection?*

We use our ILP formulation to answer this question. Our approach is to model approximation of cost estimates as a modification to the objective function of our ILP formulation. Consider the objective function $Z$ defined using accurate benefit values $b_{ik}$ and the objective function $Z^*$ resulting from substituting a benefit value $b_{ik}$ by a lower bound $b^*_{ik}$, such that $b^*_{ik} \leq b_{ik}$. Let $X_{opt}$ be the optimal solution to the original ILP instance and $Z_{opt}$ be the corresponding objective function value. Similarly, let $X^*_{opt}$ be the optimal solution to the modified ILP instance that uses the lower bound and $Z^*_{opt}$ be the corresponding objective value. Our goal is to bound the difference $Z_{opt} - Z^*_{opt}$. We follow an approach similar to that of Section 4.3.4. Consider the objective function value $Z^+$ computed by using the solution $X_{opt}$ with the modified objective function $Z^*$. Obviously

$$Z^+ \leq Z^*_{opt} \leq Z_{opt} \tag{4.22}$$

Since $Z$ and $Z^*$ differ only in the benefit value for combination $x_{ik}$, from 4.22 follows:

$$Z_{opt} - Z^*_{opt} \leq Z_{opt} - Z^+ \leq b_{ik} - b^*_{ik} \tag{4.23}$$

Equation 4.23 implies that the quality of the approximation determines the loss of benefit when optimizing with benefit lower bounds. If we apply approximations to all benefit values for all the queries, we can write:

$$Z_{opt} - Z^*_{opt} \leq \sum_{q_i} \max_{x_{ik}}(b_{ik} - b^*_{ik}) \tag{4.24}$$

Equation 4.24 does guarantee that a good quality approximation will result in a small quality loss, however we can not directly apply it because we don't know the accurate benefit values $b_{ik}$. We exploit the bound of 4.24 by replacing $b_{ik}$ by $b^i_{max}$, where $b^i_{max}$ is the maximum possible benefit achievable for query $q_i$, also used in Section 4.3.4.

Naturally we would like to exploit the ability of INUM to trade performance for accuracy. By using Equation 4.24 we can identify those queries with the largest contribution in the error bound. We can perform additional optimizer calls only for those queries so that the derived bounds are improved and improve the overall quality of the derived solution.

## 4.5 Experimental Setup

We implemented our ILP-based index selection system using Java and interfaced it to a commercial DBMS. Our implementation allows us to experiment with real-life databases and workloads and with a real commercial optimizer. Furthermore, we compare our ILP-based approach directly with the state-of-the-art index selection tool provided with the DBMS. We use CPLEX, a commercial linear programming tool to solve the resulting ILP instances. Our experiments were carried out on a dual Xeon 3GHz server.

We experiment with a workload consisting of 1000 queries, randomly selected from the TPC-H workload. The parameters for the TPC-H queries were also modified according to the *QGEN* utility and the database size was 1GB. We compare our ILP-based approach against the commercial index selection tool integrated in our server, along two

dimensions: solution quality and running time, for various amounts of storage space available for index selection.

We compute solution quality similarly to [15, 2], 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:

$$1 - \frac{c_{indexed}}{c_{unindexed}} \tag{4.25}$$

The total running time computed for our ILP-based system includes:

1. INUM setup time. The time to compute the plans in the INUM Space that are reused for cost estimation.

2. Model creation time. The time taken to perform candidate selection, combination selection and cost estimation for the resulting index combination.

3. Solution time. The time taken to solve the resulting ILP instance using CPLEX.

The total running time is compared to that of the commercial index selection tool.

## 4.6 Experimental Results

We construct the ILP instance for our input 1000 query TPC-H workload by performing candidate selection as described in Section 4.3.3 and combination selection using the table-set approach of Section 4.3.4. We used the same "interesting" table-set for all the queries, consisting of tables *LINEITEM*, *ORDERS*, *PART* and *CUSTOMER*. Table 4.1 shows the number of candidates and combinations considered by our

| | ILP | Commercial |
|---|---|---|
| Combinations | 348147 | 42189 |
| Candidates | 1931 | 138 |

Table 4.1: Number of candidates and combinations considered by our ILP-based and the commercial index selection tool.

ILP-based and the commercial index selection tool. The first column of Table 4.1 corresponds to the size of the resulting ILP instance. The data for the second column were obtained through the profiling of the commercial tool. Due to the efficiency of our ILP solver and the INUM, our ILP tool was able to handle an order of magnitude more indexes and combinations.

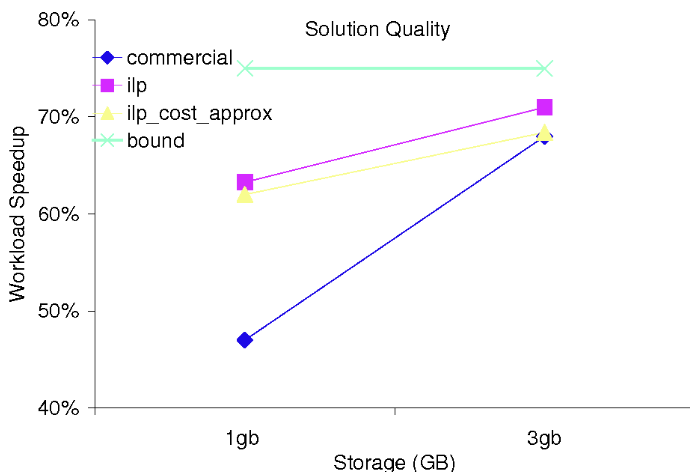


Figure 4.4: Comparing the solution quality between our ILP-based and the commercial index selection tool for two storage constraint values.

Figure 4.4 compares the solution quality achieved by our ILP-based and the commercial tool for 1GB and 3GB storage constraint values. For the "tight" storage constraint of 1GB, the ILP algorithm (ilp) provides 16 more percentage points of benefit to the workload compared to the commercial tool (commercial). Equivalently, when using the

indexes recommended by our ILP tool the workload runs 30% faster

Figure 4.4 also shows the optimal workload performance ("bound" line), obtained by optimizing this query individually without regard to the storage constraint. This bound is the same obtained by utilizing the $b_{opt}$ benefit values in Sections 4.3.4, 4.4. Achieving this bound requires an impractical amount of storage (50GB), but is easy to compute and is useful in estimating the optimality of existing approaches. The ILP formulation, using the table-subset pruning approach comes within 12% of this bound for 1GB of storage and within 4% for the 3GB case, which is an acceptable quality tradeoff given the running time superiority of our ILP approach (discussed later).

Finally, Figure 4.4 shows the quality achieved using cost approximation, in addition to table-subset pruning (ilp_cost_approx). In this case we applied cost approximation using a single plan per query in the INUM space (and thus performing only one optimizer call per query!). The quality loss resulting from cost approximation was less than 3 percentage points in both cases. As we show next, this small quality loss allowed for a dramatic reduction in running times.

Figure 4.5 compares the running times of the ILP-based and the commercial design tools, for the 1GB constraint case (the 3GB case results are identical). The running times for the ILP-based algorithm is broken down into the time to set up the cost model (INUM, see Section 4.3.2) ("inum setup"), the time to create the ilp model for the problem ("ilp setup") and the time to solve the ilp problem using cplex ('solver'). The commercial tool ran for approximately 7 hours, while our ILP algorithm using table-set pruning ran for 44 minutes, an order of magnitude improvement. The improved performance of ILP is a result of the use of INUM for cost estimation, instead of the query optimizer and of the very fast optimization engine, that took

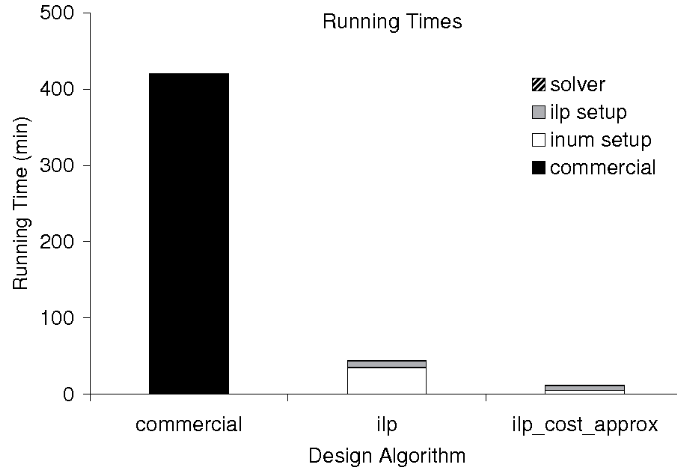


Figure 4.5: Running times for a commercial and the ILP-based design algorithms.

only 1.5 minutes to find an optimal solution. Using approximate cost estimates ('ilp_cost_approx') reduced the setup time for the INUM to only 5 minutes, making the total index selection time 12.2 minutes, a 34x speedup!

## 4.7 Conclusion

This chapter develops an Integer Linear Programming (ILP) model for the index selection problem. We apply standard optimization techniques to compute optimality bounds, derive approximate solutions with known distance for the optimal and improve approximate solutions. We describe an efficient implementation architecture that makes use of optimizer estimates similarly to commercial tools. Our experimental results indicate that ILP-based index selection is efficient efficiently and offers higher quality solutions compared to existing tools.

# Chapter 5

# AutoPart: Workload-Aware Schema Design for Large Scientific Applications

## 5.1 Introduction

Scientific experiments in fields such as astronomy and biology typically require accumulating, storing, and processing very large amounts of information. The ongoing effort to support the Sloan Digital Sky Survey (SDSS) [40, 63] provides a comprehensive example for both the terabyte-scale storage requirements and the complex workloads that will execute on future database systems. Similarly, the Large-aperture Synoptic Survey Telescope (LSST) [66] dataset is expected to be in the scale of petabytes (the data accumulation rate is calculated at 8 terabytes per night). Typical processing requirements on these datasets include decision-support queries, spatial or temporal joins, and versioning. The combination of massive datasets and demanding workloads stress every aspect of traditional query processing.

In environments of such scale, query execution performance heavily depends on the indexes and materialized views used in the underlying

physical design. The database community has recently focused on tools that utilize workload information to automatically design indexes [15, 2, 73]. Currently, all major commercial systems ship with design tools that identify access patterns in the input workload and propose an efficient mix of indexes and materialized views to speed up query execution. Typically, the tools tend to generate a set of "covering" indexes per query to enable index-only query processing (essentially, these indexes implement and ordered partition of the table). In the case of large-scale applications like SDSS, performance depends upon a large set of covering indexes, since accessing the large base tables (even through nonclustered indexes) is prohibitively expensive.

Large numbers of covering indexes are expensive to store and maintain, as data columns from the base table are replicated multiple times in the index set. Adding multiple indexes to multi-terabyte scientific databases typically increases the database size by a factor of two or three, and incurs a significant storage management overhead. In addition, indexing complicates insertions and updates. For instance, new experimental or observation data are often inserted in the database and derived data are recalculated using new models. During update operations, all "replicated" new and updated data values must be sorted and written multiple times for all the indexes. Insertion and update costs increase as a function of the number of tuples inserted or modified. If update or storage constraints do not exist, the workload can always be processed using a complete set of covering indexes. Such a scenario, however, is unrealistic for large-scale scientific databases, where both insertion and storage management costs are seriously considered.

This paper describes AutoPart, an automated tool that partitions the tables in the original database according to a representative workload. AutoPart receives as input a representative workload and designs a new schema using data partitioning. By first designing a partitioned

schema and then building indexes on the new database, queries can scan the base tables efficiently as well as a smaller set of indexes, thereby alleviating unnecessary storage and update statement overhead. Because data partitioning increases spatial locality, it improves memory and disk system performance when the covering index set cannot be built due to storage or update constraints. This paper makes the following contributions:

1. We introduce AutoPart, a data partitioning algorithm. AutoPart receives as input a representative workload and utilizes categorical and vertical partitioning as well as selective column replication to design a new high performance schema.

2. To evaluate AutoPart we build an automated schema design tool that can interface to commercial systems and utilize cost estimates from the DBMS query optimizer.

3. We experimentally evaluate AutoPart on the SDSS database and workload. Our experiments i) evaluate the performance improvements provided by partitioning alone, without the use of indexes and ii) quantify the performance benefits of partitioned schemas when indexes are introduced in the design.

Our experimental results confirm the benefits of partitioning: Even without the use of indexes, a partitioned schema can speed up query execution by almost a factor of two when compared to the original schema. Partitioning alone improves query execution performance by a factor of two on average. Combined with indexes, the new schema also outperforms the indexed original schema by 20% (for queries) and a factor of five (for updates), while using only half the original index space.

This paper is structured as follows: Section 5.2 discusses the partitioning problem in greater detail. In Section 5.3 we present the AutoPart algorithm and Sections 5.4 and 5.5 discuss the AutoPart architecture and our experimental setup. Section 5.6 presents our experimental results and Section 5.7 our conclusions.

## 5.2 Workload-based Data Partitioning

In this section, we first briefly describe the vertical partitioning idea and the factors that limits its efficiency. We then explain how categorical partitioning and replication can alleviate the problem, using examples drawn from real scientific databases.

A general formulation of the vertical partitioning problem is the following: Given a set of relations $R = R_1, R_2, ..., R_n$ and a set of queries $Q = Q_1, Q_2, ..., Q_m$ determine a set of relations $R^* \in R$ to be partitioned and generate a set of fragments $F = F_1, F_2, ..., F_N$ such that:

1. Every fragment $F_i \in F$ stores a subset of the attributes of a relation $R \in R^*$ plus an identifier column.
2. Each attribute of a relation $R \in R*$ , is contained in exactly one fragment $F_i \in F$. (except for the primary key).
3. The sum of the query costs when executed on top of the partitioned schema, $cost(Q, (R - R^*) \cup F)$ is minimized.

We expect the workload cost over the partitioned schema to be lower, because the fragments are faster to access than the original relations and queries will be able avoid accessing attributes they do not use. We define the Query Access Set (QAS) of a query $Q$ with respect to

a relation $R$ (QAS($Q, R$)), as the subset of Rs attributes referenced by $Q$. In the ideal case, where for all query pairs $Q_i, Q_j$, QAS($Q_i, R$) $\cap$ QAS($Q_j, R$)$= \emptyset$, the solution to the vertical partitioning problem would be to simply generate a fragment F for each distinct QAS in the workload. Then, each query would have to access a single fragment containing exactly the attributes it references, resulting in minimal I/O requirements. Realistically, however, the workload will contain overlapping QAS. In this case, such "clean" solutions to the vertical partitioning problem are not possible.

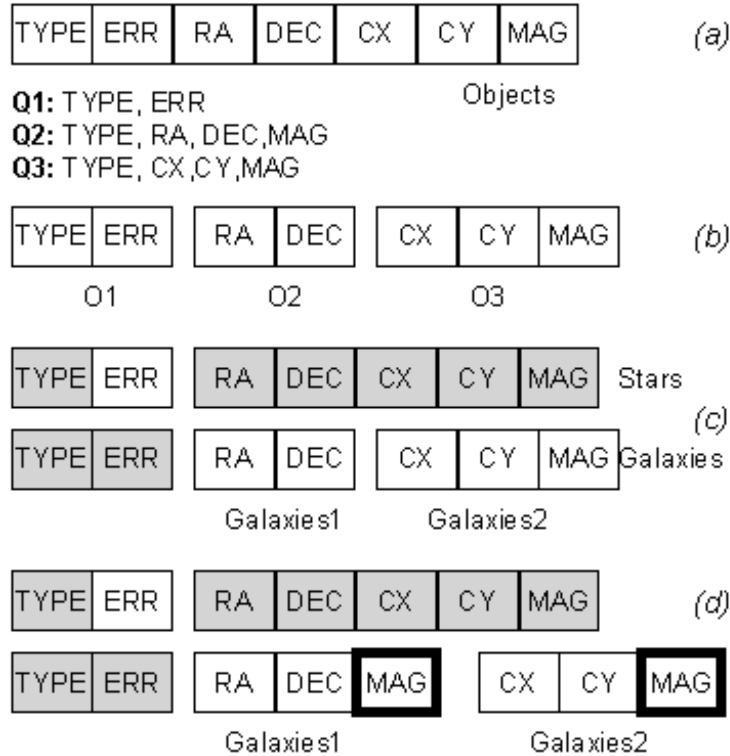


Figure 5.1: Partitioning example. (a) Original schema. (b) After vertical partitioning. (c) After vertical & categorical partitioning. (d) Column replication.

Consider the example in Figure 5.1 (a), drawn from a simplified astronomical database. Our database consists of a single table (Objects) that stores astronomical objects (galaxies and stars). Our workload consists of queries $Q_1, Q_2, Q_3$, shown in the figure with their QAS. Figure 5.1 (b) shows one possible solution for vertically partitioning

Objects into 3 fragments $(O_1, O_2, O_3)$. $Q_1$ needs to access only $O_1$, minimizing its I/O requirements. Since the attribute TYPE exists in all QAS, queries $Q_2$ and $Q_3$ will have to access fragment $O_1$ in addition to $O_2$ and $O_3$ and perform the necessary joins. Also, since $QAS(Q2) \cap QAS(Q3) = \{MAG\}$ $Q_2$ will have to access fragment $O_3$ to obtain its missing attribute, performing an additional join. Alternatively, merging some of the $O_1, O_2, O_3$ would result in lower joining overheads, but the queries would have to access a larger number of additional attributes and the I/O cost would increase.

The previous example demonstrates that overlapping QAS in a workload reduce the efficiency of vertical partitioning, because it is impossible to avoid additional joins for some of the queries in the workload. Often, however, much of the overlap implied by comparing the QAS is not real. Consider, for instance, that in the previous example $Q_1$ restricts its search to objects of type "Stars", whereas $Q_2$ and $Q_3$ only care about objects of type "Galaxies". In this case, considering only QAS leads to "false sharing" as $Q_1$ will process a completely disjoint set of tuples than $Q_2$ and $Q_3$. By categorically partitioning Objects we remove the overlap between $QAS(Q_1)$ and $QAS(Q_2) \cup QAS(Q_3)$, since they now access only the categorical fragments (Figure 5.1 (c)). Now, the fact that Q1 needs to access attributes TYPE, ERR together does not affect queries $Q_2$, $Q_3$. In addition, TYPE can be removed from the two horizontal fragments altogether. With this form of partitioning queries benefit not only from the elimination of unnecessary accesses to objects of the wrong class, but also from the removal of categorical columns. Application of categorical partitioning is the first step of the partitioning algorithm used in AutoPart.

Note that even in the categorically partitioned schema of Figure 5.1 (c), there is still an overlap between $QAS(Q_2)$ and $QAS(Q_3)$ on MAG. The impact of such overlaps, which cannot be removed by categorical

partitioning, can be reduced by allowing the replication of attributes belonging to the intersection of two or more QAS. In our example, we replicated attribute MAG in the two fragments, Galaxy1 and Galaxy2, in order to remove the remaining joins (Figure 5.1(d)). In the resulting schema all additional joins or unnecessary data accesses have been eliminated. Attribute replication is an effective way to remove the overheads introduced by overlapping QAS. To control the amount of replication introduced in the schema, we constraint the partitioning algorithm so that it uses no more than a specified amount of space for attribute replication.

## 5.3 The AutoPart Algorithm

This section describes the data partitioning algorithm used in AutoPart. The input to AutoPart is a collection of queries $Q$, a set of database relations $R$, and parameter denoting denoting the amount of storage available for attribute replication, which implicitly bounds the degree of replication allowed. The output is a set $F$ of fragments, which accelerate the execution of $Q$. This section presents an overview and details of the interesting stages of the partitioning algorithm.

### 5.3.1 Terminology

In our model, a relation $R$ is represented by a set of attributes, whereas a fragment $F$ of $R$ is represented by a subset of $R$. We distinguish between two kinds of fragments: atomic fragments are the "thinnest" possible fragments of the partitioned relations, and are accessed atomically: there are no queries that access only a subset of an atomic fragment. In addition, atomic fragments are disjoint and their union is equal to $R$. A composite fragment is constructed by the union of two

or more atomic fragments. The query extent of a fragment $F$ is the set of queries that reference it (if $F$ is atomic) or the intersection of the sets of queries that access each of its atomic components (if $F$ is composite).

### 5.3.2 Algorithm Overview

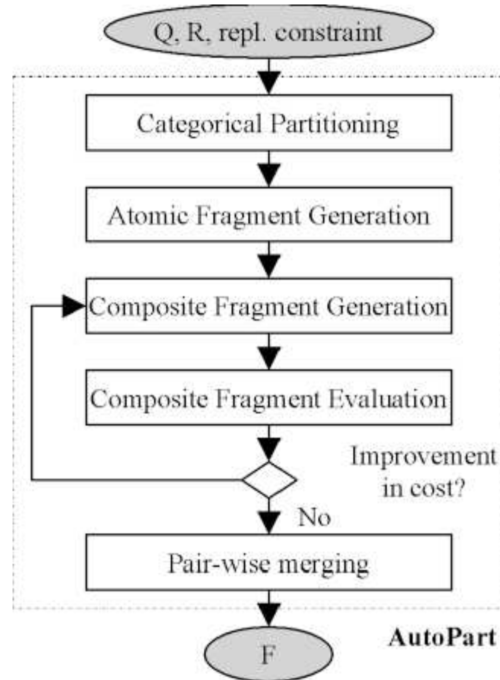


Figure 5.2: Outline of the AutoPart algorithm.

The general structure of our algorithm is shown in Figure 5.2. The first step of the algorithm is to identify the categorical predicates in $Q$, and to partition the input relations accordingly to avoid the "false sharing" between queries that have overlapping QAS but access different object classes. In the second step, the algorithm generates an initial version of the partitioned schema, consisting only of atomic fragments of the partitioned relations. The performance of this initial version of the solution is determined by the joining overhead (since atomic fragments may often contain a single attribute).

The performance of this initial schema is improved by forming com-

posite fragments that reduce the joining overhead in the resulting schema but increase I/O cost: queries accessing a composite fragment dont necessarily reference all the attributes in it. Composite fragments can either replace their constituent atomic fragments in the partitioned schema, or just be appended to the schema (assuming the replication constraint is not violated). The Composite Fragment Generation module of our algorithm determines a set of composite fragments that should be considered for inclusion in the schema, while the Composite Fragment Selection module evaluates the available options and chooses the fragments that are found to provide the highest improvements for the workload.

The algorithm iterates through the composite fragment generation and selection steps multiple times, each time expanding the fragments selected in the previous steps. The generation of fragments with an increasing number of attributes, based on the results of previous iterations, is a useful heuristic, applied also in index/materialized view selection [2, 15], to reduce the number of combination considered by the selection module. Note that the composite fragments considered may contain attributes that are also included in other fragments in the partitioned schema, thus allowing for attribute replication. When the workload cost cannot be further improved by the incorporation of composite fragments, the resulting schema is passed through a sequence of pair-wise merges of fragments, attempting to further improve performance.The following sections present the various components of the algorithm in more detail. The pseudocode for the partitioning algorithm is shown in Figure 5.3.

```
/* schema PS is the best partial solution so far */
  1. schema PS := AF
  /*Composite fragment generation*/
  2. for each composite fragment F ∈ SF(k-1)
     2.a E(f) := {composite_fragments (F,A∈AF) ∪
                  composite_fragments (F, A ∈ AF) having
                  query extent > X }
     2.b CF(k) := CF(k)∪ E(f)
  /*Composite fragment selection */
  3. for each composite fragment F ∈ CF(k)
     3.schema S_F := add_fragment (F,PS)
     3.b if size(S_F) > B then continue with the next F
     3.c compute cost(S_F, Q)
  4.select F_min = arg_max (cost (S_F, Q))
          with cost (S_Fmin,Q) < cost (PS,Q)
  5. if no solution was found then goto 9 /* exit */
  6. PS := S_Fmin
  7. SF(k) := SF(k) ∪ F_min
  8. remove F_min from CF(k)
  9. repeat steps 3-8
  /* proceed with next iteration*/
  10. k++
  11. goto 2 /*generate new fragments */
```

Figure 5.3: AutoPart pseudocode

### 5.3.3 Categorical Partitioning

The categorical partitioning step first generates horizontal fragments of the partitioned relations. The partitioning depends on the existence of categorical attributes in the relations and in the workload. Categorical attributes are attributes that take a small number of discrete values and are used to identify classes of objects. The basic motivation for categorical partitioning is that if queries operate on distinct classes of objects, those classes can be stored in separate horizontal fragments. The algorithm used for categorical partitioning of a relation $R$, under a query workload Q is shown in Figure 4. The algorithm first identifies the set of categorical attributes $A_i$ in $R$ and their corresponding domains $D_i$ (step 1). This information can be provided either by the systems

```
categorical_partitioning(relation R, queries Q, size N)
    1. A := categorical attributes ∈ R.
    2. Let X = collection of predicates in Q of the form x_i:{A_i ∈ d ⊂ D_i}
    3. XM := complete minimal set({x_i})
    4. horizontal fragments Y := minterm_fragments(X)
    5. if |Y| ≤ N then
        5a. let A := A - {a_i} /* remove attribute ai */
        5b. Yi := merge(Y,a_i) /* merge the hor. fragments defined using ai*/
        5c. Select ai that leads in the minimal |Yi|
        5d. Set Y := Yi
6. Repeat step 5 until |Y| satisfies size constraint
7. For each fragment F ∈ Y, remove all attributes in F that take a single value.
```

Figure 5.4: Categorical partitioning algorithm

.

designer or the system catalog. Each query containing predicates on those attributes, defines a horizontal subset of $R$, containing all the objects that satisfy the predicates. The purpose of the algorithm is to determine a suitable collection of non-overlapping such fragments, which will be assigned to different horizontal fragments.For this, we use the methodology developed in [13].

We can express every query predicate involving each of those attributes in the form $x_i : \{A_i \in d \subset D_i\}$. Let $X = x_i$ be the collection of such predicates, and assume that it is minimal and complete, according to [13]. Then, the min-term predicates Y(X) [13] computed in Step 4 define a collection of non-overlapping horizontal fragments that can be used to define the horizontal fragments of $R$. If there exist categorical attributes $A_i$ that take a unique value in the horizontal fragments determined, they can be removed (Step 7).

Note that this collection of fragments can be modified, for example by suitably merging the horizontal fragments determined in step 4. Such a merging is shown in steps 5a-5b. The purpose of merging could be to derive a more suitable collection of horizontal fragments In Figure 4 we restrict the number of horizontal fragments generated to a number

less than N. (This for example could express the users desire not to over-partition the horizontal schema, in order to keep its definition manageable).

### 5.3.4 Composite Fragment Generation

The composite fragment generation stage provides, in each iteration, a new set of composite fragments to be considered for inclusion in the schema. It is described by step 2 in Figure 5.3. The input to the stage for iteration k is the set of composite fragments SF(k-1) that were actually selected in the previous iteration. For the first iteration (k=1) the input to the stage is the set AF of atomic fragments.

The algorithm reduces the total number of composite fragments evaluated for inclusion by essentially extending only those fragments that were selected in the previous iteration. Those fragments can be extended in two ways:

1. By combining them with fragments in AF.
2. By combining them with fragments in SF(k-1)

The number of fragments generated in the initial steps of the algorithm is in the worst case quadratic to the number of atomic fragments. Depending on the size of the AF set, this number could be very large. It is possible to reduce the number of fragments generated, by selecting only those that will have the largest impact in the workload. Intuitively, a composite fragment is useful if it is referenced by many queries. The query extent of a fragment is a measure of a fragments importance. Step 2.a prunes the fragments that are referenced by less than X queries in the workload. Pruning based on the query extent criterion reduces the set of fragments considered during the initial steps of the algorithm. 4.5 Greedy fragment

### 5.3.5 Greedy Fragment Selection

Given the collection of composite fragments provided by the generation stage, the selection stage greedily picks a subset of those for inclusion in the partitioned schema. The selection stage is described by steps 3-8 in Figure 5.3. For each iteration, the selection module starts with the best "partial" schema found so far, PS, and a set of composite fragments CF(k) that must be evaluated for inclusion in the schema (step 3). The algorithm incorporates each candidate fragment in the current partial solution PS and computes the workload cost on the resulting schema (Steps 3.a, 3.b, 3.c). The fragment that minimizes workload cost is selected and permanently added to PS (Steps 6-8). The procedure is repeated until the workload cost cannot be further improved by fragments in CF(k).

```
procedure add_fragment (schema S, fragment F)
  for each fragment F1 ∈ S
    if (F1 ⊂ F) then remove F1 from S
      S := S ∪ { F}
    return S
```

Figure 5.5: Procedure to add fragments

The function *add_fragment* (Figure 5.5, used in step 3.a) removes all the subsets of a new fragment before adding it to the schema. This "recycling" of fragments simplifies the management of the storage space during the execution of the algorithm. If we were simply appending the new fragments to the partial solution, then the algorithm would quickly run out of space and then a separate process for removing fragments would have to be used. Using this replacement strategy , our algorithm works naturally when no replication is allowed in the partitioned schema.

```
procedure cost (workload Q, schema S)
  1.repeat for each query q in Q
  2. Let R = a relation referenced in q
  /* compute the set of partitions Q will access */
  3. P := plan (Q,S)
  4. for each fragment f in P
    4.a scan_cost := scan_cost + SR * | F|
  5. join_cost = (|P|-1) * J
  6. costR := scan_cost + join_cost
  7. repeat steps 2-6 for every relation referenced in q
  8. costQ := sum of costR + count(R) * J'
  9. return the total cost for all queries
```

Figure 5.6: The model used for cost estimation

### 5.3.6 Cost evaluation: Cost models

The selection module makes decisions based on the workload cost. We implemented AutoPart to utilize both a simple analytical cost model and the detailed cost estimation provided by the query optimizer of database systems. A simple model for the cost of a query on a partitioned schema is presented in Figure 5.6. The model captures only the parameters necessary for partitioning, like the I/O cost of scanning a table and the cost of joining two or more fragments to reconstruct a portion of the original data. In our model the I/O cost of scanning a fragment F is proportional to the number of its attributes (Step 4.a), since the number of rows in the fragments of the same relation is constant. The scaling factor SR accounts for differences in relation sizes. The cost of joining two fragments is for simplicity considered constant and equal to J. The value of J must be carefully chosen to reflect the relative cost of joining compared to performing I/O. We computed the value of J by observing the query plans generated by the query optimizer, for various partitioned schemas.Our experimentation suggests that a value for J between 5 and 10 gives good approximations of the workload cost.

An alternative to analytical models is the systems query optimizer. Modern optimizers utilize detailed knowledge of the query execution engine internals and of the data distributions to provide realistic cost estimates. The use of the query optimizer accounts for all the factors involved in query execution that our simple model ignores, like those affecting the joining costs. The use of the optimizer removes the constant join cost assumption of our model and takes into account factors like the existence of different join algorithms and the influence of predicate selectivities.The main disadvantage of using the query optimizer compared to an analytical cost model is that a call to the optimizer is time consuming.

```
procedure pairwise (queries Q, schema S)
  1. for each pair F_i, F_j in S
     1.a F_ij := merge (F_i,F_j)
     1.b S_ij := S - F_i -F_j
     1.c S_ij:= S_ij ∪ F_ij
  2. find i_min, j_min such that cost(Q,S_ij) is the minimal
  3. if cost not improved then goto 6 /*exit*/
  4. S := S i_min, j_min
  5. repeat steps 1-5
  6. return S
```

Figure 5.7: The pseudocode for pairwise merging

### 5.3.7 Pairwise Merging

The final part of the algorithm (Figure 5.7) is intended to improve the solution obtained by the greedy fragment selection through a process of pairwise merges. The algorithm merges pairs of fragments from the solution obtained so far and evaluates the impact of the merge on the workload. Merges that improve workload cost are incorporated in the solution. The loop in steps 1-5 terminates when the solution cannot be further improved. Note that merging does not increase the size of

the solution. We use the pair-wise merging process to capture the most important of those composite fragments that were not considered by the algorithm, because they were omitted by the fragment selection process.

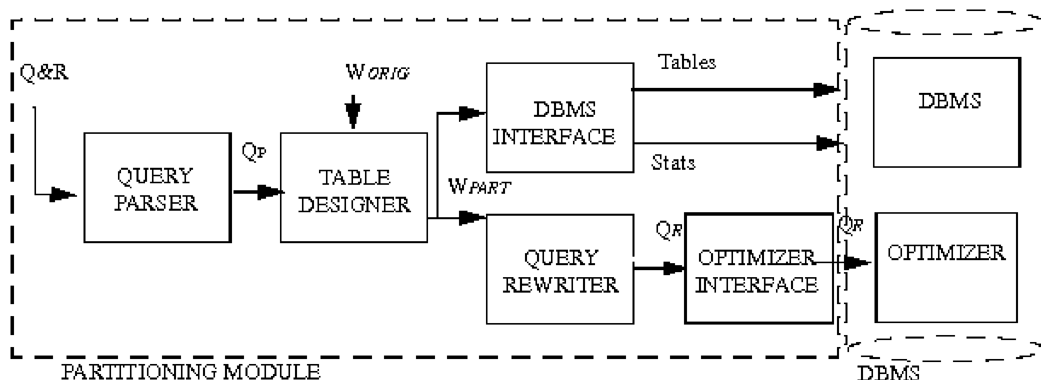

Figure 5.8: AutoPart system architecture.

## 5.4 System Architecture

This section describes the functional blocks of the automated schema partitioning tool, depicted in Figure 5.8. The system implementation was done using Java (JDK 1.4) and JDBC and the DBMS is SQL Server 2000.

**QUERY PARSER**. This module receives as input the original queries ($Q$) and the tables to partition ($R$). Its output is the queries in a parsed representation ($Q_P$)

**TABLE DESIGNER**.The Table Designer module is the heart of the schema design tool. It receives as input the set of parsed queries ($Q_P$) and the original schema definition ($W_{ORIG}$), and applies the vertical partitioning algorithms of Section 5.3. Its output is a set of candidate partitioned schemas ($W_{PART}$) to be evaluated by the query optimizer.

**QUERY REWRITER**. The rewriter uses each partitioned schema definition ($W_{PART}$) and the set of parsed queries ($Q_P$) to produce a set

of equivalent rewritten queries ($Q_R$) that can access the fragments in $W_{PART}$.

**DBMS INTERFACE**. This is a JDBC interface to the database currently hosted by the SQL Server. The interface executes table and statistics creation statements according to $W_{PART}$. To accurately estimate query costs, our tool provides the query optimizer with the correct table sizes and statistics for the partitioned schema. Since it is impractical to populate the tables for each candidate schema, we estimate table sizes and copy the estimates to the appropriate system catalog tables, for the optimizer to access. In addition, we compute statistics for each column in the original, unpartitioned tables and reuse that information for the evaluated partitions. To test our virtual table generation method, we actually implement the partitions recommended by our tool and find that the cost estimates obtained by it match those obtained from the real database.

We found that in order for the virtual and real cost estimates to agree, the statistics must be generated using full data scans and not by random sampling.

**SYSTEM CATALOG**. The DBMS catalog stores information like table sizes, row sizes and statistics. To facilitate query cost estimation, we update the system catalog tables with information reflecting the new schemas.

**OPTIMIZER INTERFACE**. This JDBC interface receives as input the rewritten queries ($Q_R$) and uses the query optimizer to obtain query plan information and cost estimates.

We deployed our partitioning tool as a web application, that runs independently of the database server component. We provide the input (query workload and tables to be partitioned) through a simple web interface. Our tool can (through standard JDBC) access remote

databases to obtain the original schemas, modify their structure and obtain cost estimates for alternative solutions.

## 5.5 Experimental Setup

Our experiments use the Sloan Digital Sky Survey (SDSS) database [40, 63], running on SQL Server 2000. The database is structured around a central "catalog" table, *PHOTOOBJ* (22GB), which describes each astronomical object using 369 mostly numerical attributes. The second largest table is *NEIGHBORS* (5GB), which is used to store spatial relationships between neighboring objects. It essentially contains pairs of references to neighboring *PHOTOOBJ* objects and additional attributes, such as distance. Both tables are clustered on their primary key, which consists from application-specific object identifiers.

The SDSS workload consists of 35 SQL queries. Most of them are sequential scans that process *PHOTOOBJ* and apply predicates to identify collections of astronomical objects of interest. 6 queries (the most expensive ones) have a spatial flavor, joining *PHOTOOBJ* with *NEIGHBORS*. Only 68 of the 369 attributes in the *PHOTOOBJ* table and 5 out of the 8 attributes in NEIGHBORS are actually referenced in the workload. For a fair comparison, we modified the database tables before our experiments, so that they only contain the attributes actually referenced in the workload.

To realistically evaluate the full impact of data partitioning one needs to include maintenance operations in the workload. The update workload (SDSS_U) used in our experiments consists of two insertion statements (*SQL INSERT*), that simulate the insertion of new data in the systems two largest tables. The statements we use simply append 800,000 and 5,000,000 tuples in the *PHOTOOBJ* and *NEIGHBORS*

tables respectively, corresponding to 6% and 4.5% of their current contents.

The SDSS database comprises 39 tables. We used our partitioning algorithm to partition the two largest ones, *PHOTOOBJ* and *NEIGHBORS*, that are almost exclusively responsible for the workloads I/O costs. We present our performance results in terms of the estimated execution time provided by the query optimizer. The speedup of a query is defined as

$$s = 1 - \frac{\text{query_cost_optimized}}{\text{query_cost_original}}$$

## 5.6 Experimental Results

In this section we present experimental results on (a) the performance of our data partitioning algorithm and (b) the benefits of partitioning in the presence of indexes and maintenance workloads.

### 5.6.1 Evaluation of Partitioning

This section demonstrates that the combination of categorical partitioning and attribute replication can generate schemas that can significantly improve query execution, even without the use of any indexes. We derive two partitioned schemas, CVP_x0 and CVP_x0.5 through categorical and vertical partitioning, without and with replication respectively. In the attribute replication case, we set a storage upper bound for the replication columns equal to 1/2 the original database size.

The SDSS queries are categorized into two groups. The first group, SDSS_J, consists of four queries, whose execution is bounded by ex-

pensive joins among several instances of *PHOTOOBJ* and *NEIGHBORS*. These queries account for 47% of the total workload cost. The second group, SDSS_S, includes 31 SDSS queries, dominated by table scans. Queries in the SDSS_J group do not benefit much from partitioning, since the joins are their dominant operators. On the other hand, we expect vertical partitioning to significantly improve performance of queries in the SDSS_S group.

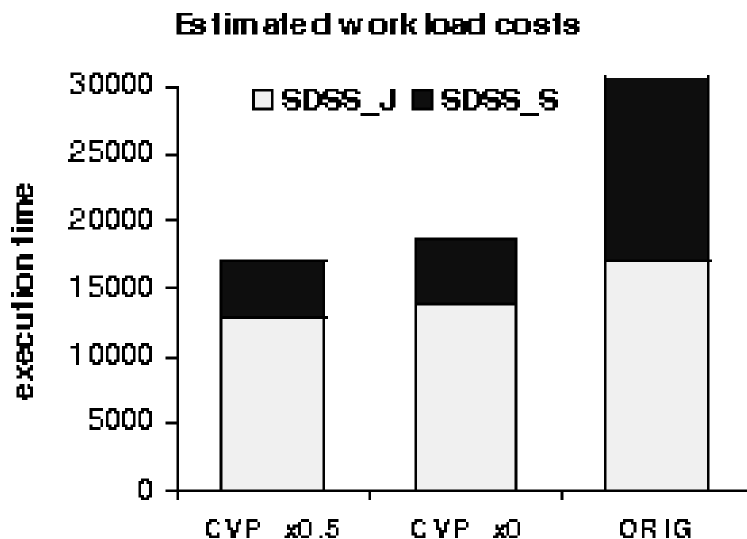


Figure 5.9: Comparison of workload costs for the two partitioned schemas and the original, for the two workload classes.

Figure 5.9 shows the estimated workload performance distinguishing and the two query classes, SDSS_S and SDSS_J. As expected the replicated schema (CVP_x0.5) performs better than the one without replication. (CVP_x0) The overall performance improvement is 47% and 43% respectively. Queries in the SDSS_J class benefit less, 19% and 24% respectively, while the improvements for the SDSS_S class queries are 69% and 72%. We observe that attribute replication after partitioning did not make significant difference in the overall execution time (8%).

Figure 5.10 shows normalized execution times for queries in the SDSS_J (left) and in the SDSS_S (right) groups. When compared to

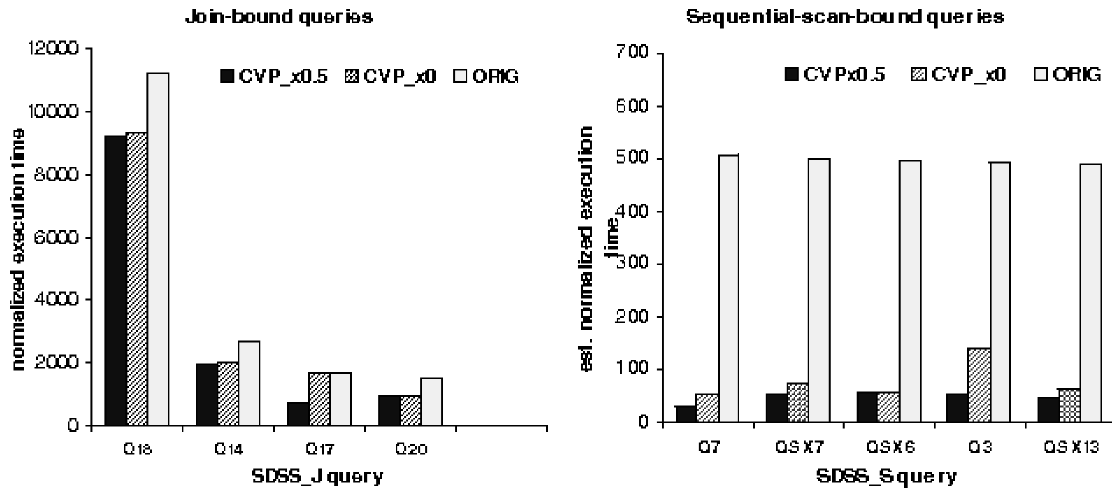


Figure 5.10: Individual query execution times for the original and partitioned schemas.

ORIG, query performance in the SDSS_J group, improves from 2% (Q17, CVP_x0) to 56% (Q17, CVP_x0.5). The performance improvement for queries in the SDSS_S group is often impressive (an order of magnitude for Q7).

### 5.6.2 Indexing a Partitioned Schema

This section shows the benefits of partitioning even when compared to an unpartitioned schema with indexes. We designed indexes using the Index Tuning Wizard in SQL Server 2000. We allowed unlimited storage for indexes, but we added updates (SDSS_U) to the input workload. The cost of the SDSS_U workload increases considerably with every new index built, since that index would require the updated data to be properly ordered. Since the partitioned schemas are already optimized for the particular workload, they will require much less indexing effort, offering better performance for both retrieval and update statements

Figure 5.12 shows the total workload cost when using the indexed original and partitioned schemas, for all the statement groups (SDSS_J,

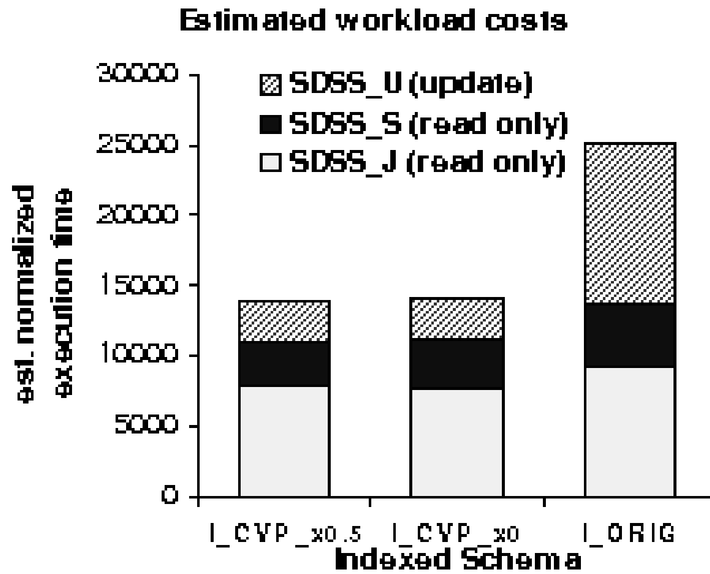


Figure 5.11: Query and update workload costs using the original and the partitioned schemas.

SDSS_S, and SDSS_U). When using the I_CVP_x0 and I_CVP_x0.5 schemas, read-only statements run 20% faster compared to the original schema, whereas the insertion statements are more than 5 times faster. Overall, the partitioning improves query execution performance even in the presence of indexes, by approximately 45%.

Figure 5.12 shows the total amount of storage allocated for the I_ORIG and the two partitioned schemas, broken down into the storage required to index the two main tables. The partitioned schemas require about half the storage space for indexes, compared to the original schema. According to Figure 5.12, the original schema relies on heavily indexing *PHOTOOBJ* for performance. In comparison, because of the performance benefits of partitioning *PHOTOOBJ*, the partitioned schemas require 7 and 4 times less storage of indexing. Instead of heavily indexing *PHOTOOBJ*, the partitioned schemas allocate some more space for the efficient indexing of *NEIGHBORS*.

Our experimental results in this section show that partitioning can improve query execution performance, requiring less indexing overhead.

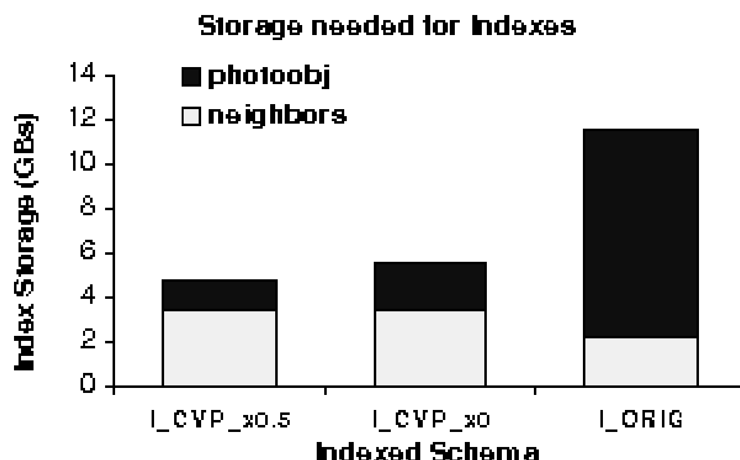


Figure 5.12: Index storage when using the original and the partitioned schemas.

## 5.7 Conclusions

Database applications that use multi-terabyte datasets are becoming increasingly important for scientific fields such as astronomy and biology. In such environments, physical database design is a challenge that involves complex query processing needs as well as space limitations. We propose AutoPart, an algorithm that automatically partitions database tables utilizing prior knowledge of a representative workload. Using a data partitioning and replication, Auto- Part suggests an alternative, high-performance schema that executes queries faster than the original one and can be indexed using a fraction of the space required for indexing the original schema. To evaluate AutoPart, we build an automated schema design tool that interfaces to commercial database systems. The paper describes our algorithm, the system architecture, and experimental results using the Sloan Digital Sky Survey database.

# Chapter 6

# Efficient Query Processing on Unstructured Tetrahedral Meshes

## 6.1 Introduction

Simulations are crucial for studying complex natural phenomena, from the flow of hot gas inside a propellant to the propagation of cracks inside materials, earthquakes and climate evolution. Recent advances in modern hardware allow scientists to carry out simulations of unprecedented resolution and scale. Accurate simulations improve our understanding and intuition about complex physical processes, but in order to reap their benefits we must be able to search for useful information in the haystack of large-scale simulation output datasets.

### 6.1.1 Querying Simulation Datasets

To analyze and display simulation results, post-processing and visualization applications query a discretized version of the application domain (typically represented using a *mesh*) and the simulation output. Figure 6.1 shows the architecture of *Hercules*, a simulation application developed by the Quake group at Carnegie Mellon [4, 70, 47], that

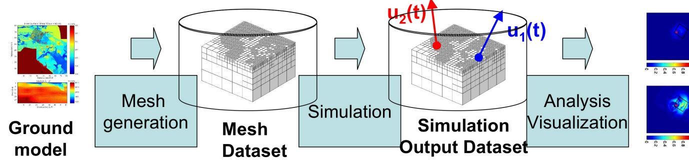


Figure 6.1: An earthquake simulation pipeline.

computes earthquake propagation for a given ground region and initial conditions. The simulation receives as a grid-like discrete ground model and at each simulated time-step it computes the ground velocity at the mesh points, storing the result in the *simulation output*. In *Quake* simulations, mesh models typically consume hundreds of gigabytes and simulation output volumes are in the terabyte scale [4]. To fully utilize the information involved in a modern simulation, post-processing and visualization applications need efficient, scalable query processing capabilities.

To organize the data representing the discretized application domain, simulations typically employ an *unstructured tetrahedral mesh*. A tetrahedral mesh models the problem domain by decomposing it into tetrahedral shapes called *elements*. For instance, Figure 6.2(a) shows a part of a mechanical component mesh model, whereas Figure 6.2(b) illustrates a constituent tetrahedral element. The element endpoints, called the *nodes*, are the discrete points on which the simulation computes physical parameter values, like ground velocity. Tetrahedral elements have varying sizes, angles and orientations. When dealing with complex geometries that require variable resolution, the tetrahedral mesh is a powerful modeling tool due to its unique flexibility in defining arbitrarily-shaped elements. Therefore, tetrahedral meshes are vital for a wide range of applications in areas like mechanical engineering and earthquake modeling.

The most frequent query types on tetrahedral meshes are spatial,

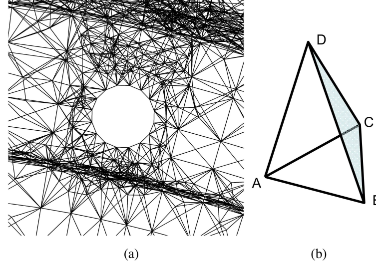


Figure 6.2: (a) Part of a tetrahedral mesh dataset modeling a mechanical component. (b) A tetrahedral (pyramid) mesh element and its four endpoints, the nodes.

point and range queries. They retrieve the mesh elements intersecting an input query region or containing the query points, along with the corresponding mesh nodes. Such queries are important in visualization or analysis applications that interpolate the values of physical parameters (like ground velocity) at a given point or a region, from the values computed at the mesh nodes. We also identify a class of queries called *feature* queries, that retrieve arbitrarily shaped regions of the dataset that are important for the application, such as surfaces and boundaries.

Query processing performance is critical for the scientific processing applications that require interactive rendering rates (less than 1s per frame [72]). High frame rates are impossible to achieve on large-scale mesh datasets without efficient indexing techniques. Unfortunately, the pyramid-based geometry of tetrahedral meshes, while increasing their expressive power, makes developing effective indexing methods a challenging task. To tame the long execution times, scientific applications typically compromise accuracy either by utilizing a subset of the mesh data or by resorting to less flexible structures. Because meshes are difficult to index and query efficiently using spatial indexing methods available in today's Database Management Systems (DBMS), applications use specialized programs instead. As the size and complexity of the datasets grows, however, these programs suffer from scalability, performance, and portability limitations [35].

### 6.1.2 Our Approach and Contributions

In this dissertation, we introduce *Directed Local Search (DLS)*, a query processing approach for tetrahedral mesh datasets. DLS avoids the complexities involved in trying to capture the geometry of the mesh, by utilizing the *connectivity* between mesh elements.

DLS uses a novel application of the Hilbert curve to obtain an initial approximate solution, which is "refined" through local search algorithms. Our technique relies on the distance preserving properties of the Hilbert curve and on an efficient representation of connectivity information to provide significantly better performance compared to traditional techniques that rely only on geometric approximation.

DLS allows the construction of simulation applications that can efficiently query large-scale meshes stored in a database system along with implementation simplicity and easy integration with existing DBMS.

The detailed contributions of this dissertation are:

1. This is the first study to treat query processing on tetrahedral mesh data, a crucial problem for large scale scientific applications, using database technology.
2. We evaluate and compare the performance of the prevailing spatial indexing methods when applied on tetrahedral meshes, explaining their inefficiencies.
3. We design Directed Local Search (DLS), an efficient algorithm for indexing and querying large unstructured tetrahedral meshes. To index a mesh efficiently, DLS for the first time:
   (a) Combines mesh topology information with the mesh geometry.
   (b) Applies the Hilbert space-filling curve for approximate indexing

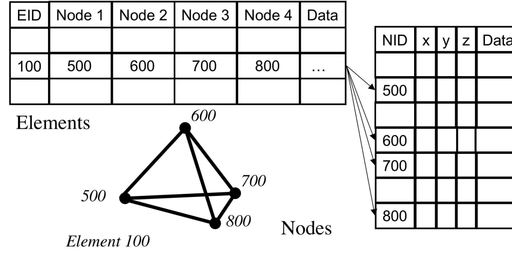


Figure 6.3: The mesh representation in the database consists of separate tables for the mesh elements and nodes.

4. We implement DLS in a simple and efficient fashion, using standard access methods (the ubiquitous B-Tree). In addition, we use graph-based techniques to efficiently store dataset topology information.
5. Experiments with DLS running on top of PostgreSQL show that DLS results in a reduction in the number of I/O accesses and query execution time by 25% up to a factor of 4.
6. We propose a graph-based technique for clustering mesh elements on disk pages and we show that it improves the I/O performance of *feature* queries by 16% to 37.9% compared to traditional linear ordering based on the Hilbert space-filling curves.

This chapter is structured as follows. Section 6.2 details the database design and query workloads. Section 6.3 evaluates the prevailing spatial indexing techniques on tetrahedral mesh datasets. In Section 6.4 we describe Directed Local Search and the related indexing and data organization and in Section 6.5 our element clustering approach. Section 6.6 details our implementation and experimental setup, while Section 6.7 presents our experimental results. We conclude with Section 6.8.

## 6.2 Background

We use the database organization shown in Figure 6.3. The mesh components, elements (tetrahedra) and nodes (points), are stored in separate tables. Each *Elements* record contains the IDs of the 4 corresponding nodes, while *Nodes* holds the coordinates for each node. This organization is suitable for spatial queries, as it allows fast access to all the nodes of an element. There exist other relational mappings for meshes [35], but they are tuned towards different query types, for instance determining all the elements sharing a given node.

We consider the following 3 types of queries:

1. A point query simply returns the containing element (and its nodes). Post-processing and visualization applications use point queries in order to interpolate the value of a physical parameter on the particular query point, given the values computed by the simulation at the nodes. This general functionality is vital to virtually every application that requires values at points that do not coincide with the input mesh node set.

2. Range queries return a set of elements contained in or overlapping with the rectangular query region, along with the corresponding nodes. A range query is used to retrieve "chunks" of data that are then fed to a (possibly parallel) visualization or analysis tool.

3. *Feature* queries return regions of the dataset that have arbitrary shapes. Features in datasets are defined by the application. For example, in earthquake analysis the ground surface is a feature of particular interest if we want to measure earthquake impact on buildings.

## 6.3 Traditional Indexing on Tetrahedral Meshes

Database literature provides a wealth of multidimensional indexing techniques. Gaede et al. provide an excellent survey on the topic [25]. In this section we demonstrate that existing techniques have sub-optimal performance for tetrahedral meshes and/or exhibit low storage utilization and preprocessing overheads.

R-Tree-based approaches approximate objects by their Minimum Bounding Rectangles (MBRs) and index them with an R-Tree [31] variant. Performance optimizations involve packing, clipping and replicating overlapping MBRs and using more complex bounding shapes (polyhedra). We investigate the applicability of R-Tree based techniques in sections 6.3.1 and 6.3.2.

Another approach is to overlay a rectilinear grid over the indexed domain and approximate each object by one or more grid cells. The cells are arranged and indexed using coordinate transformations like the Z-order. Z-order based techniques are investigated in section 6.3.3.

### 6.3.1 R-Tree Based Techniques

The R-Tree search for a query starts from the root level and follows a path of internal nodes whose MBR intersects the query range or contains the query point. If multiple nodes at one level match the query criteria the search will follow all possible paths, requiring more page accesses. There exists a large body of research on improving performance by minimizing the area of R-Tree nodes and the overlaps that lead to multiple paths. Dynamic techniques like the original R-Tree construction algorithm [31] and the R*-Tree [6] maintain an optimized tree structure in the presence of data updates. Static techniques like the

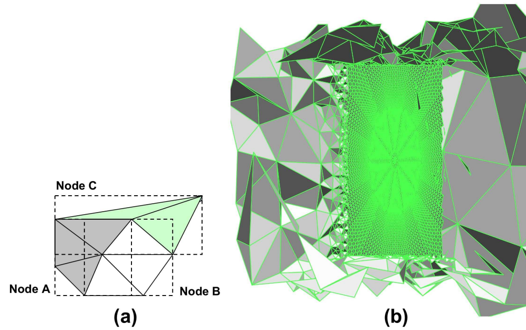


Figure 6.4: (a) Overlaps between the leaf-level nodes of an R-Tree for a 2D mesh. (b) The mesh used in the R-Tree experiments.

Hilbert-packed R-Tree [41], the Priority R-Tree [5] and others [57, 21] attempt to compute an optimal R-Tree organization for datasets that do not change.

Tetrahedral meshes are a challenging application for R-Trees because they have many overlapping MBRs. Figure 6.4 (a) shows a two-dimensional example. The overlap between the R-Tree leaf nodes A and B is significant and it is not clear how to arrange the individual triangles to minimize it. Also, the elongated triangles in the upper right part give a very large surface to node C.

We evaluate R-Tree performance with a real dataset used for crack propagation simulations [35]. We compare two dynamic implementations, the originally proposed quadratic-split R-Tree and the R*-Tree and two packed implementations, the Hilbert-packed R-Tree and the STR-packed R-Tree [44]. The first two are implemented using PostgreSQL, and the others are taken from the "Spatial Index Library" ([33]).

Figure 6.4 (b) shows part of our dataset structure. Our mesh models a mechanical component with a crack at its center. It contains a very high-resolution central area, surrounded by more coarse-grained elements.

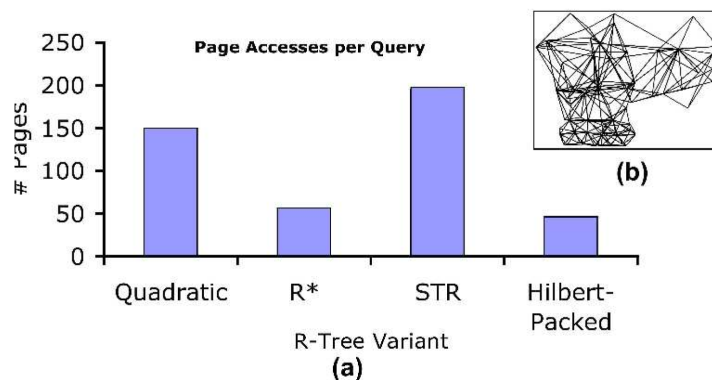


Figure 6.5: (a) Average number of page accesses per point query for 4 R-Tree variants, where all trees have 3 levels. (b) Internal structure of a leaf-level node of the Hilbert-packed R-Tree.

Figure 6.5 (a) shows the average number of page accesses for the 4 indexes over 1000 point queries focused on the dense region. The indexes perform 30-198 page accesses, while the tree height (and hence the theoretical minimum number of accesses) is 3. Figure 6.5 illustrates the cause for the measured performance, showing one of the Hilbert-packed R-Tree's leaf nodes. The node has an unnecessarily large volume because of the large elements on the top, that 'cover' the smaller elements. Any query intersecting the bottom right part of the MBR (which is approximately where the dense region of the dataset is) will have to unnecessarily hit that node.

Extensions like the P-Tree [39] attempt to improve R-Tree performance by using Minimum Bounding Polyhedra. The P-Tree relies on polyhedra with faces aligned to a fixed set of $d$ orientations. Such 'constrained' polyhedra will likely still lead to overlaps because they do not capture the geometry of every pyramid in the mesh. In addition, they require additional storage for storing the more complex bounding approximations.

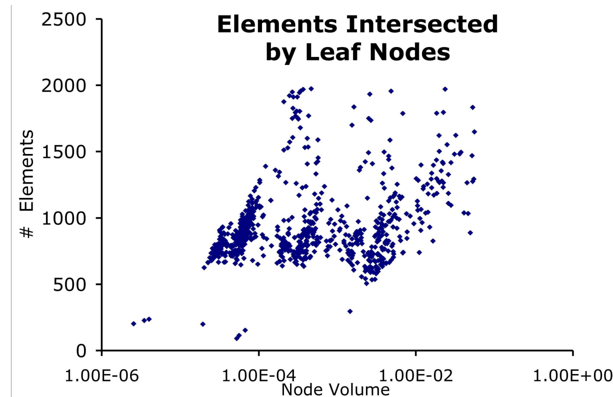


Figure 6.6: Number of elements intersected by R-Tree nodes, as a function of node volume.

### 6.3.2 Clipping

A clever solution to the problem of overlaps is to use non-overlapping regions. The R+-Tree [59] ensures non-overlapping tree nodes by generating disjoint MBRs during node-splitting and *replicating* the objects that cross MBR boundaries.

The R+-Tree attempts to improve performance by trading search efficiency with higher storage overhead, as pointers to the same object consume space in more than one tree nodes. The increased storage requirements of the R+-Tree make it undesirable for scientific datasets: Simulation datasets are challenging exactly because of their unprecedented volumes and it makes no sense to adopt an indexing solution that multiplies the storage needed for each object!

Storage space is not the only problem: reduced storage utilization negatively affects performance. Unlike point que-ries, that benefit from the non-overlapping nodes, range queries will suffer because links to objects within the query range will be retrieved multiple times. The performance of loading data in the index will also be problematic, because of the more complicated loading algorithm that also needs to write a lot more data).

| Grid Resolution | Number of intersected elements |
|---|---|
| 16x16x16 | 100 |
| 32x32x32 | 40 |
| 64x64x64 | 20 |
| 128x128x128 | 10 |

Table 6.1: Number of mesh elements intersected by grid cells with varying resolution.

We highlight the R+-Tree inefficiency by showing how indexing a tetrahedral mesh dataset requires an unreasonable amount of replication. We measure the number of tetrahedra intersected by leaf-level nodes of different sizes, for the mesh dataset described in Section 6.3.1, using the nodes generated by the Hilbert R-Tree of Section 6.3.1 as a guide. Figure 6.6 shows the number of mesh elements intersected by nodes as a function of the MBR's volume. According to Figure 6.6, 80% of the nodes intersect 500 to 1000 elements, which will have to be replicated. Given that each node contains 120 entries, the new dataset would require 5 to 10 times more space.

More sophisticated clipping-based techniques, like the disjoint convex polygons of the cell-tree [30] exhibit similar inefficiencies: First, the cell-tree construction algorithm still requires the replication of objects that cross partition boundaries, like in the R+-Tree case. Furthermore, the space overhead of keeping the polygon descriptions in the tree nodes is much higher compared to that of storing MBRs and leads to poor storage utilization for large datasets.

### 6.3.3 Z-Order based techniques

Z-order based techniques overlay a rectilinear grid on the indexed domain. Each object is approximated by a collection of grid cells and a linear cell ordering is computed using the Z-order [53, 54].

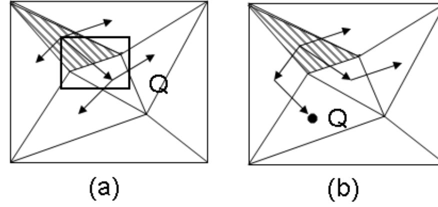


Figure 6.7: (a) A 2D example of Directed Local Search. (b) Directed Local Search for a point query.

The cells intersecting the query range (or the cell containing a query point) is identified by comparing the query's Z-order value range with those of the cells, using a B-Tree. Query performance depends on the grid resolution. A very fine grid will lead to a large number of indexed cells, but each cell will overlap with only a few elements. Higher resolution reduces the number of cells at the cost of losing precision and performance.

Investigating the tradeoffs involved in picking the right resolution and thus minimizing the impact of replication is an interesting exercise. However, regardless of the optimal resolution, the z-order approach will suffer from the same inherent problems of replication, described in Section 6.3.2.

Table 6.1 demonstrates the amount of required replication for varying grid resolutions, for the uniform, dense region of the dataset in Section 6.3.1. Figure 6.1 shows the average number of tetrahedra overlapping a cell. For the high resolution grid, which consists of $128^3$ cells, the dataset requires $128^3 \times 10 = 20971520$ records, which implies a $20\times$ increase in the dataset size. Lower resolutions reduce the storage overhead, at the expense of search time (100 elements must be searched for the $16 \times 16 \times 16$ decomposition).

```
1. Identify a suitable starting element E
2. enqueue (BFS_Queue, E); mark E as visited
3. While (BFS_Queue not empty)
   3.1 element e = dequeue(BFS_Queue)
   3.2 if e intersects Q
     3.2.1 add e to the query result
   3.3 for each ni in neighbors(e)
     3.3.1 if face fi intersects Q
             and ni not visited
     3.3.2  enqueue (BFS_Queue, ni)
     3.3.3  mark ni as visited
```

Figure 6.8: The Directed Local Search algorithm

## 6.4 Directed Local Search

In this section we present Directed Local Search (DLS). We first present the basic DLS algorithm for range and point queries and then describe in detail the techniques that enable DLS, namely the proximity search algorithm and the compressed representation of mesh connectivity information.

### 6.4.1 Algorithm Overview

Directed Local Search processes range and point queries by utilizing the mesh connectivity. Figure 6.7 (a) shows an example range query on a 2D triangular mesh. If we know a single initial element that is part of the answer (highlighted) we compute the query result by searching first all of the initial element's neighbors and incrementally expanding the search in a breadth-first search (BFS) fashion until we find no additional elements within the range. Figure 6.7 (b) shows the same principle applied to a point query: Starting at an initial element, we perform the same expansion until we reach the target element.

Figure 6.8 details the DLS algorithm for a range query $Q$. Step 1 identifies a starting element that intersects the range query, using the *proximity search* algorithm described in the next section. The next steps describe the breadth-first search (BFS), that stops when no further

elements intersecting the range can be found. The predicate in step 3.3.1 examines if the face $f_i$ of the current element $e$ intersects the query range. If it does not, it is not necessary to visit neighbor $n_i$.

The primary advantage of DLS over traditional spatial indexing is that the breadth-first search is independent of the mesh geometry. Using the mesh connectivity avoids the performance problems generated by overlapping MBRs. Furthermore, DLS does not approximate tetrahedra by simpler shapes and therefore directly computes the query results instead of first forming approximate answers and then post-processing them.

Implementing DLS requires solving the following subproblems. First, we need a way to determine a suitable starting element (step 1 of Figure 6.8) that intersects the query range. We call this starting element selection "proximity search" and describe it in Section 6.4.2. For point queries, the proximity search described in the next section directly finds the containing element and thus the BFS search of Figure 6.8 is not used. In addition, we need to efficiently represent the "neighbor" relationships between elements, so that BFS can quickly access them. In Section 6.4.3 we show how to improve over the adjacency-list representation of connectivity information.

Finally, note that DLS is guaranteed to succeed if the dataset is convex, not containing any holes or concavities. This is the case for many models used in practice (ground, materials models). In practice, our techniques work also for small holes or concavities because the BFS can "work around" them as long as they are not too big. A more general treatment involves cataloguing all the exterior mesh surfaces and "jumping" from one surface face to another as long as they are contained in the query region. Such a solution is part of our ongoing work.

| Query | Point | Range 1% | Range 5% | Range 10% |
|---|---|---|---|---|
| "hit" %' | 52% | 93.3% | 94.7% | 97.7% |

Table 6.2: BFS "hit rates" for various query types

### 6.4.2 Proximity Search

In this section we present algorithms for selecting an initial "seed" element for DLS. We use the Hilbert curve to index the tetrahedral elements, after representing each element by its center point. For a range query, we find the element whose center has the closest Hilbert value to the center of the query range. For a point query, we identify the element closest to the query point in Hilbert space (Figure 6.9 (a)). We work similarly for range queries, looking for an element close to the center of the query range. The distance preserving properties of the Hilbert curve [50] imply that the obtained element will be close to the query region and thus a suitable starting point.

The Hilbert ordering translates the proximity search, a geometric operation, into a numerical one. Thus we are able to use linear access methods (the B-Tree) that are fast and predictable, avoiding the complexity and cost of accessing a spatial access method.

To our knowledge this is the first time that the Hilbert curve is directly used for spatial indexing. (Multiple Hilbert curves have been proposed for nearest neighbor queries [45]). The problem is that there exist no guarantees about the distance between the returned element and the query point. The novelty of our approach is that using the connectivity information, we can correct the problem by reaching a suitable starting element even if the Hilbert index returns an element that is far away.

Figure 6.9 (b) outlines our basic proximity search algorithm, Hilbert_BFS. Hilbert_BFS selects an initial element using the Hilbert value index and

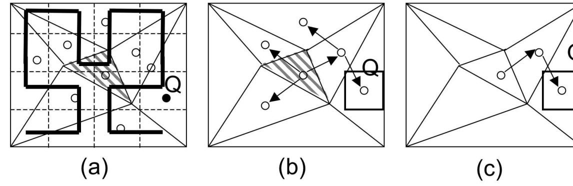


Figure 6.9: (a) Using the Hilbert order to select a starting element for point Q. (b) Hilbert_BFS example. (c) Hilbert_Direct example.

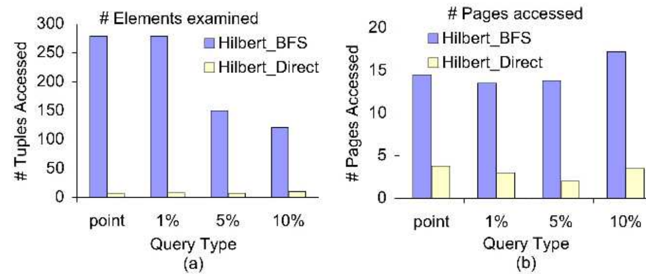


Figure 6.10: (a) Elements examined by the BES variants. (b) Corresponding page accesses

if it doesn't overlap the query region, it performs a breadth-first expansion until another suitable element is found. Regardless of the initial element the algorithm will eventually return an overlapping element, as in the worst case the entire dataset will be scanned. The Hilbert ordering of the elements benefits BFS by increasing spatial locality and minimizing page accesses.

Figure 6.9 (c) presents Hilbert_Direct, an improvement over the basic Hilbert_BFS algorithm. Instead of "expanding" the search towards all directions, Hilbert_Direct follows a path of elements towards the center of the query region. The next neighbor in the path is determined by "drawing" a line connecting the center of the current triangle to the center of the query region and crossing the face intersected by the line. Hilbert_Direct resembles a depth-first search, since it expands only the one neighbor that lies in the direction of the query region. It is still useful to remember the other neighbors as well, since following alternative paths makes the algorithm robust to small concavities and "holes".

We now present a characterization of the above algorithms in terms of their effectiveness in determining a suitable starting element, using our *gear* dataset discussed in Section 6.6. We answer the following questions:

1. How often does the Hilbert index provide a suitable starting element immediately? For range queries, we measure the percentage of queries that immediately find an intersected element. For point queries, we measure the percentage of queries where the returned element contains the query point.
2. How many elements does each of the above techniques examine before a suitable element is returned?
3. How many *page accesses* does each of the above search techniques require?

Table 6.2 shows how effective the Hilbert index would be if it was used by itself. The Hilbert index immediately returned the correct element for half of the point queries. Also, more than 90% of the queries can be answered by the Hilbert index directly. Thus Hilbert indexing by itself is highly efficient, returning immediately suitable results most of the time. Similar results were obtained for all the datasets described in our experimental section.

Figures 6.10 (a)-(b) characterize the performance of Hilbert-_BFS and Hilbert_Direct, by measuring the tuples and pages accessed until we reach a suitable starting element, considering only the queries that *were not answered immediately by the Hilbert index*. We consider point queries and range queries whose size is 1, 5 and 10 percent of the dataset size. Hilbert_BFS accesses a much higher number of tuples than Hilbert_Direct, because it searches towards all directions. Since

Hilbert_Direct offers the best performance, we use it in our implementation and experimental evaluation.

Hilbert_Direct uses the same geometric principles as the "triangulation walking" studies [22] by the computational geometry community, where the main focus is the theoretical analysis of point query performance. A complete theoretical analysis is a separate area of study, where the additional assumption of uniformly distributed mesh nodes is necessary for tractability [23] (such a theoretical analysis is beyond the scope of this dissertation). When running point queries, at most 128 elements were processed by a single query before finding a suitable starting point, incurring 9 page accesses. In our experiments, only one in a thousand queries exhibits this worst-case scenario, hence the excellent average performance shown in Figure 6.10.

### 6.4.3 Representing Element Adjacency

In order to implement the algorithms of the previous section we need to be able to retrieve the neighbors for each mesh element. A simple way to obtain this connectivity information is through the mesh generation process itself: Mesh generators like *Pyramid* [60] can provide the neighbors of each tetrahedral element as part of the output. If the connectivity information is not accessible, there are simple techniques to compute it, by using a hash table to match the elements with the same face. There is even a way to compute element connectivity using a standard commercial DBMS, as outlined in [37] [1]. The development of optimized ways to extract connectivity information from large meshes is part of our ongoing work.

The connectivity information comprises, for each element, the location on the disk of its 4 (at most) neighbors. A disk pointer is a

[1] By generalizing their surface extraction algorithm.

(*page_no*,*offset*) tuple ID. The simplest way to store the pointers is to extend the *Elements* table with 4 additional columns. However, it is desirable to develop a more efficient representation technique for the connectivity information in order to improve the storage requirements and I/O performance of our solution.

We propose a compressed representation based on the clustering properties of the Hilbert curve. By computing the Hilbert ordering for the elements, besides facilitating the efficient proximity search of Section 6.4.2, we *re-label* the elements so that the spatially close elements receive IDs that tend to be numerically close. The implication is that the neighbors of an element are also likely to receive similar IDs. We take advantage of this labeling property by actually storing the *differences* between the IDs of an element and its neighbors. The motivation is that in the common case the difference will be much smaller than the IDs themselves and thus, with an appropriate encoding scheme, it will require fewer bits. Figure 6.11 shows an example. The neighbors of element "1000" received similar IDs. The ID differences are orders of magnitude smaller and require fewer bits to represent.

We now describe our compression scheme in more detail. Given an element with ID $E$ and 4 neighbors with IDs $E_1$, $E_2$, $E_3$, $E_4$, we encode them by storing the values $code(E - E_1)$, $code(E - E_2)$, $code(E - E_3)$, $code(E - E_4)$ in a variable length field. For the compression to be efficient, the integer encoding function $code(.)$ must represent small values with fewer bits compared to larger values.

We use *snip* codes [7], shown to provide both high storage efficiency and performance for general purpose graph compression. The snip encoding of an integer is the actual binary representation of the integer, in a linked list format. Each 2-bit "snip" contains 1 binary digit from the number's representation and one "continue" bit that is set to zero

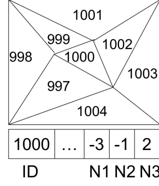


Figure 6.11: 2D example of adjacency compression. Logical ID differences require fewer bits.

| Dataset | Rows | Bits/Record | Fragmentation |
|---|---|---|---|
| gear | 8.8M | 14.5 | 3% |
| circle | 10M | 14.3 | 3.1% |
| cube | 5.4M | 14.8 | 3.4% |
| heart | 510K | 12.2 | 2.9% |
| quake | 14M | 14.2 | 3.3% |

Table 6.3: Compression results for our real datasets.

only for the last snip. We use one additional snip for signs, as ID differences could be negative and we prefix the code with 2 bits denoting the neighbor count.

In practice, instead of logical IDs, we need to store the differences of page numbers (encoding the offset within the page is easy, as it is typically small). Compressing page differences is not straightforward. We need to know the page numbers of neighboring elements in advance, but this is impossible as the page number of an element depends in turn on the compression of all the previous elements!

For simplicity, instead of developing complex multi-pass mapping algorithms we use a constant page capacity $P$ for the entire dataset. We set $P$ to be equal to the minimum page capacity when we use logical ID encoding. Since the page number differences are smaller than the ID differences, this approach might generate some free space in the page. We can minimize the free space by increasing $P$ an recomputing the differences, up to the point where we do not cause page overflows.

Table 6.3 quantifies the compression achievable for our experimen-

tal datasets and the amount of internal fragmentation incurred and characterizes the storage overhead of our technique. The connectivity information requires fewer than 2 bytes per record even for large datasets, in contrast to the 24 bytes of the adjacency-list implementation. The small overhead (increased space utilization) translates to improved I/O performance for range queries.

### 6.4.4 DLS Generalization

DLS can form the basis for a more general indexing tool that can support other types of mesh shapes besides the tetrahedra, such as bricks, prisms, or pyramids with more faces (like pentahedra). DLS is extensible because for all of the above shapes we can exploit the foundations of our technique:

1. There exists a Hilbert encoding, that allows for proximity search and clustering.
2. We can easily evaluate point containment or range intersection predicates.
3. We can exploit element connectivity.

Extending DLS to handle *finite volume* meshes, such as those used in Computational Fluid Dynamics (CFD) is not as straightforward, because such meshes are often represented in a face-oriented rather than element-oriented fashion. The absence of explicit elements means that are no constraints in the volumes enclosed by faces, allowing non-convexities and multiple elements sharing a face. Furthermore, there are numerical tolerance issues in determining whether a face contains a query point. A solution to this problem requires the development of new indexing techniques and is also part of our ongoing work.

## 6.5 Graph-based clustering for Tetrahedral Mesh Data

Our techniques use the Hilbert curve to cluster mesh elements on the disk, minimizing the *Elements* page accesses. In this section we show that, while Hilbert clustering offers very good performance for rectangular, box-shaped range queries, it is sub-optimal for queries on arbitrarily shaped regions (like surfaces) that are frequent on scientific applications.

To improve the clustering for these cases, we introduce the idea of graph-based clustering. Rather than relying on element center coordinates, graph based techniques try to place an element on the same page with its neighboring elements, as frequently as possible.

Our approach goes beyond space-filling curve clustering, so far the only general-purpose layout technique for spatial data, by allowing efficient retrieval of arbitrary, application-specific regions *without* sacrificing the overall spatial locality of the layout (and thus without requiring multiple copies or orderings of the same data and without affecting DLS indexing).

### 6.5.1 Graph Partitioning and I/O

We use the *dual graph* representation of a mesh, mapping each mesh element to a graph vertex and each pair of neighboring elements to an edge. Using the dual graph, we restate the abstract problem of "preserving element spatial locality" as a graph partitioning problem: We partition the dual graph vertices into page-size chunks so that we minimize the number of edges crossing page boundaries. This formulation implies spatial locality because, intuitively, nodes connected by an edge are likely to be retrieved together by a range query. .

Graph partitioning is a hard problem [26] but there exist many practical heuristics. We use METIS [43], a multi-level graph partitioning heuristic, shown to offer the best known results [7]. The dual representation of a mesh is given by the mesh generation process and thus we can use METIS directly. In the case of very large models, we can first coarsely partition them into memory-sized chunks using the Hilbert clustering and use METIS on each individual partition. Alternatively, we could modify METIS to produce memory-sized first level partitions.

### 6.5.2 Feature-Based Clustering

Scientific datasets commonly have distinct *features*, connected regions repeatedly queried by the application. An example is the ground surface in earthquake simulations, as we are interested in the damage inflicted on the buildings. Features are usually known in advance and are heavily queried because we need to access the relevant data *for every simulated time step*.

Hilbert curve clustering is suboptimal for querying features, because it is optimized for rectangular, box-like queries. It is well known that its quality deteriorates with increasing query region hyper-surface [50]. We use graph-based clustering to improve the retrieval performance for feature queries. Our approach is based on *explicitly* specifying frequently co-accessed elements and on using this information to guide data layout. The dual graph of the mesh dataset is ideal for this purpose, as co-access information can be encoded into the edge weight.

As the example in Figure 6.12 shows, we overlay the dual graph on the feature region and strengthen the weights for the edges within the region. Our experimental results suggest that it is sufficient to increase the edge weights so that the total weight associated with the region is is comparable to the total original weight of the entire graph. If the

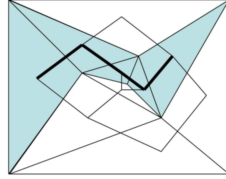

Figure 6.12: A 2D feature example (highlighted)

weight increase is too small, it will not affect the overall quality of the solution. Due to the increased edge weights, graph partitioning will pack the feature's elements in disk pages. sacrificing some of the edges not fully contained in the feature. The remaining "regular" edges help by still maintaining the overall spatial locality in the dataset.

Note that feature based clustering is not related to indexing: The feature specification and the assignment of elements to features is application dependent. Rather than identifying the feature elements, feature-based clustering reduces the I/O cost of *retrieving* them after they have been identified by the application. Also, although we are motivated by datasets queried repeatedly (per time-step), in this dissertation we only consider spatial queries (not involving multiple time-steps).

## 6.6 Experimental Setup

In this section we describe the techniques, datasets and methodology used to experimentally evaluate DLS.

### 6.6.1 Implementation

We implemented DLS on top of PostgreSQL (version 7.4.5). The data is stored in the *Elements* and *Nodes* tables as shown in Section 6.2. We store element center coordinates using the *cube* datatype included with

| Name | Elements | Nodes | Size (GB) | R-Tree levels | B-Tree levels |
|---|---|---|---|---|---|
| gear | 8.8M | 1.3M | 1.3 | 4 | 4 |
| circle | 10M | 1.5M | 1.4 | 4 | 4 |
| cube | 5.4M | 0.9K | 0.6 | 4 | 4 |
| heart | 570K | 110K | 0.1 | 3 | 3 |
| quake | 14M | 2.5M | 2 | 4 | 4 |

Table 6.4: Datasets used in our experiments.

the PostgreSQL distribution and use the center for sorting and building the B-Tree for the proximity search, based on the Hilbert ordering. To incorporate the Hilbert order, we replaced the PostgreSQL comparison routines with code that compares Hilbert values *directly* from IEEE double precision coordinates, without actually computing them. The direct comparison routines allow us to use the highest possible Hilbert curve resolution, equivalent to 192-bit Hilbert values. The additional storage required for the connectivity information is shown in Table 6.3 of Section 6.4.3. The DLS routines were implemented as new join operators, based on the PostgreSQL nested loop join.

We compare DLS to a Hilbert R-Tree implementation also built on top of PostgreSQL. We utilized the GiST access method, which we modified to allow for the Hilbert R-Tree bulk loading method. The Hilbert R-Tree is optimized so that it stops the search once the containing element for a point query is found, eliminating unnecessary page accesses.

Our experiments run on a 2-way P4 (3.6 GHz, 2MB L2) Xeon machine with 4GBs of memory and 2 320 SCSI-2 hard drives, running Linux 2.6.

| | |
|---|---|
| rtree | HRTree pages accesses. |
| btree | B-Tree pages accesses (DLS). |
| elements | The *Elements* page accesses. |
| nodes | Accesses to *Nodes* and its index. |

Table 6.5: PostgreSQL page access categories.

| | |
|---|---|
| rtree | The time for an R-Tree lookup. |
| btree | The time for a B-Tree lookup. |
| nodes | Time for a lookup on the *Nodes* table. |
| elements | Time for DLS proximity search and BFS. |
| others | Containment predicate, direction computation. |

Table 6.6: Running time breakdowns.

### 6.6.2 Datasets and Queries

We experiment with the real 3D mesh datasets shown in Table 6.4. The *gear*, *disk* and *cube* meshes are used in crack propagation whereas the *quake* meshes are used in earthquake simulations. Finally, the *heart* dataset is a model of a human heart developed for use in biomedical applications[2].

Our query workloads consist of uniformly distributed point and range queries. We vary the sizes of the range queries, so that the range size is equal to 1%, 5% and 10% of the dataset size.

### 6.6.3 Performance Metrics

We report page accesses and query running times for the point and range queries described in the previous sections. For page access counts, we report the number of *distinct* database pages accessed per query, broken down into the subcategories of Table 6.5. We measure running times on a "cold" system, where no data or index pages are cached in main memory at the beginning of each query. We use cold measurements because they better capture the impact of I/O on query

[2] Made available by the Computational Visualization Center at the University of Texas, Austin (http://ccvweb.csres.utexas.edu/cvc/).

performance. We break the running times into the components shown in Table 6.6.

Performance improvements in terms of *speedups* of DLS over the Hilbert R-Tree (HRTree) are computed by:

$1 - Page\ accesses\ (DLS)/Page\ accesses\ (HRTree)$ and
$1 - Running\ Time(DLS)/Running\ Time(HRTree)$.

Constructing the Hilbert index is essentially a standard sorting operation that uses the comparison routine of Section 6.6.1. It is handled by the DBMS and we therefore do not report its performance. Besides, the Hilbert R-Tree which we compare against has exactly the same performance. The connectivity information is computed and stored during the mesh creation time and involves an additional pass over the *Elements* table for computing the new element IDs after sorting the elements according to the Hilbert order.

## 6.7 Experimental Results

In this section we experiment with DLS and compare its performance against the Hilbert-packed Tree (HRTree).

### 6.7.1 Range Query Performance

Figures 6.13 (a)-(e) show the average number of page accesses for range intersection queries of varying sizes, on 5 datasets. For each pair of bars, the first corresponds to the HRTree (labeled rtree) and the second to DLS (dls). As Figure 6.13 demonstrates, the R-Tree access is the largest component in all the cases (up to 74% for *circle* 1%) except for

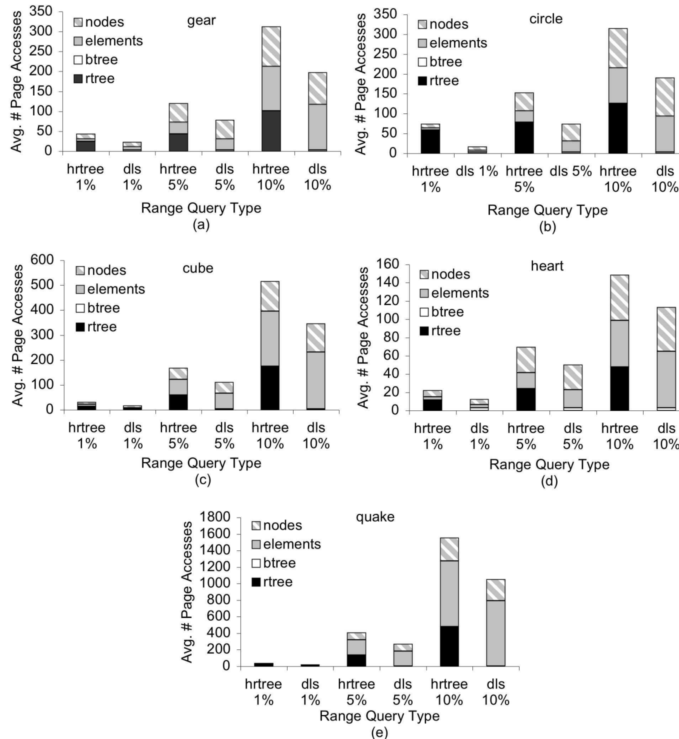


Figure 6.13: Page accesses for range queries of varying sizes on our 5 datasets for the Hilbert R-Tree and DLS.

the "large" 10% queries where it is comparable to the *Elements* table accesses. For the 10% queries, the R-Tree is responsible for 32% (*heart*) to 39% (*circle*) of the accesses.

DLS eliminates the R-Tree overhead by combining an efficient B-Tree lookup and a localized proximity search instead of a costly R-Tree traversal operation. The B-Tree lookup requires 4 page accesses for all datasets (except for *heart* with only 3 levels). The proximity search is highly efficient, requiring 0.5-2 additional page accesses on average across all datasets. The overall effect is a reduction of up to 96 times (*quake* 10%) in the number of page accesses required for indexing.

The improvement is larger for our more complex datasets (*gear*, *circle*, *quake*) as opposed to the more uniform ones (*cube*, *heart*), highlighting the robustness of DLS with respect to the geometric complexity of the dataset (the same trend appears in the results for point queries, Sec-

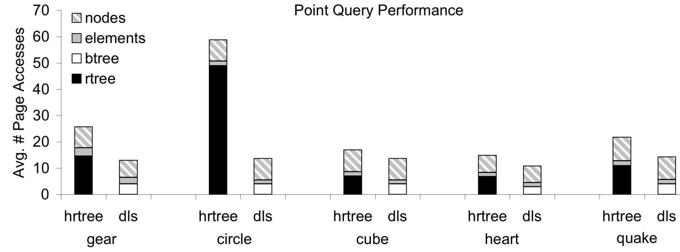


Figure 6.14: Page Accesses per point query for the Hilbert R-Tree and DLS on our 5 datasets.

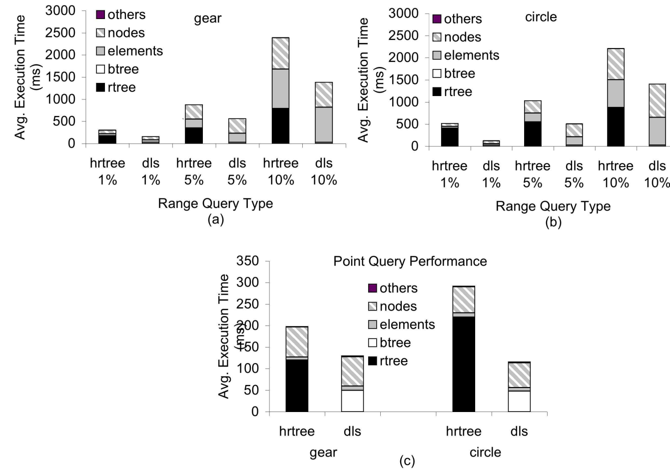


Figure 6.15: (a)-(b) query execution times for range queries on *gear* and *circle*. (c) Query execution times for point queries.

tion 6.7.2). The *cube* and *heart* meshes contain more regular elements (in terms of size/geometry), characteristics that help the performance of the HRTree. The *gear*, *circle* and *quake* datasets are representative of the real meshes used in applications: In practice only certain regions of the domain are "refined" (modeled with large numbers of tiny elements), which leads to significant irregularities in the mesh structure.

The number of *Elements* pages accessed by DLS is comparable to that of the HRTree, even for the 10% queries. Due to the effectiveness of our compression technique, the additional connectivity information does not deteriorate I/O performance. For all the "small" ranges (1%) DLS accesses 1-3 fewer pages than the HRTree, corresponding to a

reduction of up to 48%. This happens because the R-Tree accesses leaf pages whose MBR intersects the query range, but do not contain a result. The impact of this imprecision decreases for larger ranges, as it is more likely that a leaf page will actually contain an element intersecting the query range and is anyway needed by the query.

The overall performance improvement offered by DLS ranges from 28% to up to a significant factor of 4 (*circle*, 1%). The *circle* dataset benefits from DLS the most, with improvements of 36% up to a factor of 4.

Figure 6.15 (a), (b) shows the query execution times for the same workloads on the *gear* and *circle* datasets. The query execution times confirm the trend in the page access count results. DLS can improve overall query execution performance by up to a factor of 4 for *circle* 10%.

### 6.7.2 Point Query Performance

Figure 6.14 shows the average number of page accesses for point queries. The results are similar to the range query results. R-Tree accesses correspond to 41% (*cube*) to 83% (*circle*) of the total page accesses. As in Section 6.7.1, *circle* and *gear*, the most irregular datasets in this study, have the highest number of R-Tree accesses.

DLS replaces the expensive R-Tree lookups with B-Tree page accesses that need only 4 page accesses (3 for the smaller *heart*). The elimination of the R-Tree leads to overall improvements ranging from 19% up to a factor of 4.

The number of *Elements* pages accessed by the R-Tree and DLS methods is very similar, 1.67-2.5 for the HRTree and 1.4-1.64 for DLS. DLS accesses slightly fewer pages on average, as the R-Tree might have

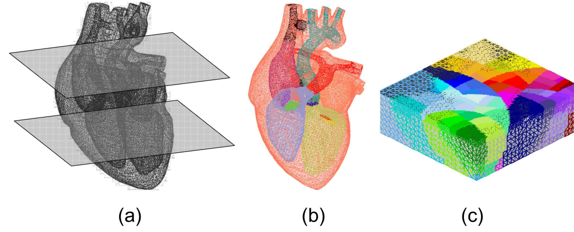


Figure 6.16: (a) Heart model with cross-sections. (b) Heart model with boundaries. (c) Partitioned ground mesh

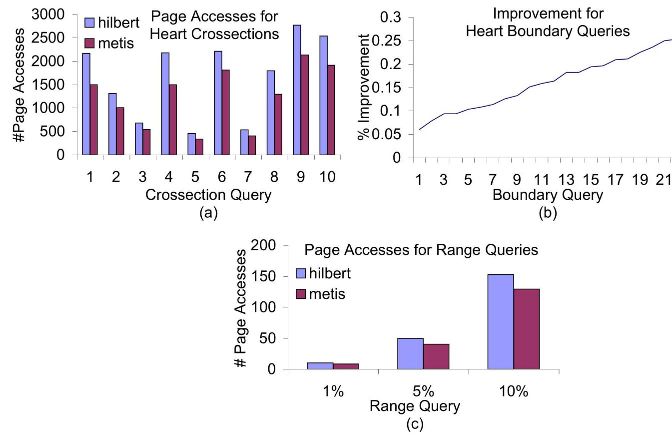


Figure 6.17: (a) Page accesses for retrieving the 10 cross-sections. (b) % Improvement for retrieving boundaries. (c) Average accesses for range queries.

to access more elements simply because their MBRs intersect the query point, without actually being part of the solution. The Hilbert clustering of *Elements* however helps in keeping those elements on the same page.

Figure 6.15 (c) shows the query response times for point queries on two datasets, *gear* and *circle*. The execution times confirm our page count measurements and demonstrate that the reduction of R-Tree page accesses by DLS can lead to significant savings in the overall query response time. Similar to our page access results, *gear* and *circle* show the largest running time improvements, 34% up to a factor of 2.5.

### 6.7.3 Feature Clustering: Heart Model

In this section we show that feature-based clustering provides better I/O performance compared to Hilbert curve clustering for feature queries. We use two feature examples on the *heart* dataset.

Figure 6.16 (a) shows our first experiment, a situation where several cross-sections of the heart model have been identified in advance and are used for querying the model.

We use 10 such cross-sections, each represented by a "thin" range query, with height equal to 1% of the dataset height, randomly spread within the model. The elements intersected by each cross-section are known in advance and our goal is to reduce the number of page accesses required to retrieve them.

Figure 6.17 (a) compares the performance of Hilbert-order clustering to that obtained by using the ideas in Section 6.5.2 and the METIS partitioning tool. Feature-based clustering reduces the number of pages accessed per cross-section by 18%-31%, for an average improvement of 25%. More importantly, as Figure 6.17 (c) demonstrates, this performance improvement comes at no cost for the average spatial range query performance. In fact, the layout obtained through graph partitioning improves range queries by 15%-19%.

For this particular scenario the Hilbert curve clustering could be modified to favor accesses along horizontal cross-sections, by appropriately "stretching" the Hilbert cells. This approach however does not work for arbitrarily shaped features. As the next example illustrates, feature-based clustering is a natural match for such datasets.

Figure 6.16(b) shows the heart model augmented with additional surfaces corresponding to various heart components (like the pulmonary valve or the aorta). This information is provided by the model's con-

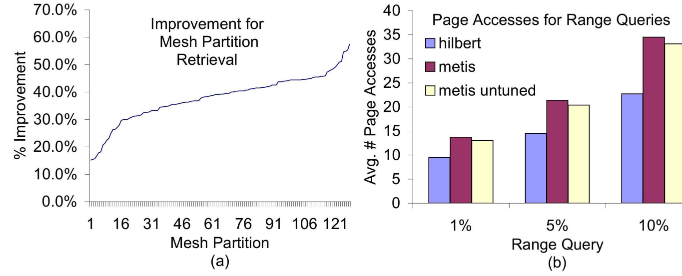


Figure 6.18: (a) % Reduction in the number of page accesses per partition for the earthquake dataset. (b) Average page accesses for range queries

structor by assigning each *node* to either one of 22 different *boundaries* or to the model's interior. Using the node information we identify the elements adjacent to each boundary and treat each boundary as a separate feature.

Figure 6.17 (b) compares the page accesses per surface for the Hilbert and feature-based clustering. The improvements obtained by our layout are 6% to 25% for an average of 16%. Again, like in the previous example. there is no impact on the performance of the random range query (Not shown).

### 6.7.4 Feature Clustering: Earthquake model

Figure 6.16 (c) shows a ground model mesh used in earthquake simulations. The mesh is partitioned into 128 parts for parallel simulation on 128 computing nodes. Improving the performance of retrieving the individual mesh components is useful for applications that move the same data between the storage subsystem and the computation nodes multiple times, like for example large-scale I/O-bound visualization systems [72] that read the same partitions for every time-step and distribute them to rendering processors working in parallel.

Figure 6.18 (a) shows the reduction in the number of page accesses

obtained by feature-based clustering over the Hilbert clustering for the 128 partitions. The improvements range from 15.2% to 57.5% with an average improvement of 37.9%.

Figure 6.18 (b) shows the impact of the graph-based clustering on the performance of random range queries of varying sizes. Contrary to the previous examples, feature based clustering in this case hurts random query performance. This happens because Hilbert clustering has much better performance compared to graph based techniques for this particular dataset, even when we use graph partitioning on the initial dual graph *without* changing any edge weights (as the bar labeled *metis_untuned* of Figure 6.18 (b) demonstrates).

We believe that this happens because the mesh model we used is small (150K elements) and regularly structured. It is derived by triangulating an oct-tree mesh, thus the elements fit nicely into cubical regions and the Hilbert curve does a better job at clustering them. This example, besides highlighting the potential for improving Hilbert-based clustering, motivates further research on the general properties of graph-based partitioning, specifically on how it relates to different dataset geometries.

## 6.8 Conclusion

This dissertation examines database support for efficient query execution on large tetrahedral mesh datasets. We present Directed Local Search (DLS), a query processing technique for spatial queries that takes advantage of the mesh connectivity and its efficiency is independent of the complexity of the mesh geometry. We show that DLS can be easily implemented in a database system without requiring the development of new access methods. We also propose a new graph-based

technique for clustering mesh elements to disk pages and demonstrate that it has better performance than traditional clustering techniques using space-filling curves when retrieving regions of arbitrary shapes.

# Chapter 7

# Conclusion

“To find interesting events, physicists may apply range conditions on a handful of attributes [...]. Efficiently answering these partial-range queries is a serious challenge. The traditional indexing techniques, such as B-trees and hashing, are inefficient for datasets with a large number of searchable attributes. Even multidimensional indexing techniques, such as R-trees, [...] a brute-force scan is more efficient than these indexing schemes.”

*The Office of Science Data-Management Challenge.*
*Report from the DOE Office of Science Data-Management Workshops*
*March-May 2004.*

“Based upon the SDSS experience, if one has a careful processing plan and a good [i.e, workload-based] database design, one can create a small “summary” catalog, which will cover most of the scientific uses, the rest of the data needs to be looked at much less frequently.”

*Szalay et.al., “Petabyte-scale Data Mining: Dream or Reality?” [62]*

The two quotes above are indicative of the challenges faced by scientists when dealing with large volumes of complex datasets. The source

of the first quote, the "Data Management Challenge" was published by the D.O.E [66] and specifies a comprehensive agenda for scientific data management research. The document also distinguishes between simulation and observation-driven applications. The distinction matches the structure of this dissertation, that provides database support for simulation applications that introduce new data organizations and also for applications that directly match commercial, relational database management systems.

## 7.1 Support for New Data Organizations

For simulation-driven applications, the challenge lies in supporting the data mining and visualization tools used to process the simulation output, that involves multidimensional data in the terabyte scale. The example mentioned in the D.O.E document is the three-dimensional hydrodynamics simulations performed by the DOE SciDAC TeraScale Supernova Initiative, which produce data at the rate of 5TB/day. This dissertation develops similar examples from earthquake simulation applications [4, 70, 68, 69] and general finite element analysis applications [35, 37, 36].

The challenge shared by all applications is supporting multidimensional queries on complex data, such as finite element meshes, with very large data volumes. Existing database techniques, when applied to such new data organizations, are often worse than the simple approach of simply scanning the data. The reason is that existing multidimensional indexing applications (such as the R-Tree) were originally developed for CAD or geographic applications that have a simple structure and admit approximations (such as Minimum Bounding Rectangles). When applied to complex structures and geometries, "generic" multidimensional indexing techniques do not provide adequate performance. It

is therefore necessary to develop novel spatial indexing techniques for complex scientific data.

There exist specialized, custom-developed data management solutions [48, 71], that aim to account for the lack of general-purpose data management techniques. Such techniques have limited portability, as they are tightly integrated to the rest of the application they were developed for. Due to this lack of portability, there exist no comparative performance studies and essentially no reuse of previously proposed solutions and implementations. There is work in progress aiming to add indexing support to scientific file formats such as HDF [64]. A problem with this approach is the lack of a general model for HDF data, which would be used to guide the indexing. Furthermore, any such indexing tool would have to be separately deployed and maintained, in addition to other data management infrastructure (relational systems, for instance). Finally, developing specialized indexing techniques will eventually have to "duplicate" the mechanisms and implementation integrated in today's highly optimized database engines.

We advocate the approach of *integrating* data into existing data management solutions. The idea is to reuse the core infrastructure provided by existing database systems, such as: B-Tree indexing, join operators and parallel data access, as building blocks for managing the complex new data organizations introduced by scientific applications. An example of this approach is the Hierarchical Triangular Mesh (HTM) [29], used for spatial indexing in relational databases and the Finite Element Analysis work in [35, 36, 37].

This Ph.D dissertation contributes Directed Local Search (DLS), a multidimensional indexing technique for unstructured tetrahedral meshes, a data organization typically used by a large class of scientific simulation applications. DLS is novel in that it does not rely in geometry approx-

imations, like existing multidimensional indexing techniques. Instead of geometry it relies on *mesh topology* and on an efficient *approximate* spatial search algorithm. The combination makes DLS insensitive to the complex geometry of meshes and provides up to 4 times faster performance (in our experiments with real-world meshes) compared to optimized traditional database techniques. Furthermore, we extend the idea of relying on mesh topology for query processing. We propose a new technique for clustering mesh elements in disk pages that uses graph partitioning algorithms and can outperform traditional multidimensional clustering approaches based on space-filling curves by up to 38%.

DLS does not require the development of new, exotic data structures. The only data structure required is the B-Tree, optimized implementations of which already exist in all database systems. For this reason, DLS can be very easily integrated into existing, commercial DBMS and enable them to efficiently handle mesh data. DLS avoids the portability and compatibility problems faced by custom indexing approaches: Using DLS, the same data management infrastructure (a commercial DBMS) can be used to store both relational and complex spatial data. DLS is in fact implemented in the data management system implemented by the Cornell Fracture Group for managing FEA data [35, 36, 37].

## 7.2 Automated Database Design for Large-Scale Scientific Databases

Scientific applications dealing with observation data, such as astronomy, benefit most from existing commercial relational technology: Applications with relatively “flat” data such as catalogs of astronomical

objects, can easily be hosted by commercial DBMS and take advantage of the performance, streamlined deployment, extensibility and portability. The Sloan Digital Sky Survey (SDSS) is an example of a successful large-scale application hosted by commercial DBMS, demonstrating that DBMS technology can in principle scale to terabytes of data.

We focus on the performance of such large-scale relational deployments. Scientific query processing in relational systems naturally translates to SQL processing. SQL statements can be arbitrarily complex, involving combinations ("joins") of multiple tables, along with complex predicates, sorting and aggregation computation. In addition, databases such as SDSS that can be queried through the Web, need to process several thousands of different SQL statements, each with different requirements. The challenge in relational deployments does not involve new low-level indexing and query processing techniques, but optimizing performance in the presence of large workloads of complex SQL queries.

Improving the performance of SQL workloads is not unique to scientific applications and therefore commercial systems already ship with *automated database design tools*, that analyze input workloads and determine appropriate database structures, such as tables, indexes and materialized views [15, 2, 3, 73, 32]. Scientific applications impose two broad requirements on database design tools. The first is *solution quality*. The resulting database design must significantly improve workload performance. Ideally, workload performance will be *as good as possible* (optimal) under limited resources (such as storage space). The second requirement is *scalability*: Design tools should be able to process very large workloads, consisting of thousands of SQL statements and at the same time explore a large number of alternative designs, in order to reach good quality solutions.

The above requirements are particularly important for large-scale scientific applications, because they typically involve large workloads (coming from the Web, for instance) consisting of "expensive" statements that require good database designs for acceptable performance. Furthermore, resources are limited: Allocating 20TBs of storage to indexes for a 10TB database is probably unacceptable for scientific deployments with limited budgets. Our work on AutoPart (Chapter 5 of this dissertation) is a good example of how workload-based database design can improve both query performance and the consumption of resources for scientific databases. AutoPart introduces table partitioning in database physical design: Partitioning is an interesting design option, because it improves query I/O while requiring negligible additional storage. AutoPart performs *automated* partitioning, using the query workload to guide the partitioning. We experimented with AutoPart and the SDSS database, using a real representative SQL workload. Our experiment demonstrated that adding the partitioning option to database design improves query performance, while allowing for five times faster update performance and reducing index storage by 50%.

This dissertation contributes a new framework for automated database design that significantly improves the state of the art in terms of the quality and running time requirements discussed above. Chapter 4 discusses the cost model used by automated design tool (the query optimizer) and identifies the performance of the query optimizer as a major bottleneck in the scalability of database design tools. Chapter 4 introduces the INdex Usage Model (INUM) a framework that allows us to "cache" previous query optimizer output and reuse it to compute new query estimates "on-the-fly" without further optimizer invocation. INUM is based on the observation that although the index selection algorithm evaluates a large number of candidate configurations, the optimal plan for a given query does not necessarily change from one

configuration to the next. INUM captures exactly the conditions that cause the transition from one optimal plan to the other, when the selection of indexes changes. By eliminating the optimizer overhead, INUM offers significantly improved scalability allowing candidate sets with thousands of indexes and simpler enumeration algorithms.

Chapter 5 discusses the algorithmic approaches taken by database design tools and starts with the observation that they use heuristic approaches, that do not provide optimal solutions or known bounds from the optimal. Since the effect of the heuristics proposed in the literature has not been formally analyzed, current approaches miss the opportunity of *selectively* and aggressively applying approximations to improve scalability, without a significant impact in solution quality. This dissertation develops a radically different approach to database physical design. We model design problems using an Integer Linear Programming (ILP) formulation, that allows us to exploit existing, industry-strength optimization engines to efficiently handle very large problem instances. We also exploit the mathematical structure of the ILP formulation to analyze application-specific heuristics. Unlike previous techniques, we are able to prove bounds on the effect of our heuristics on solution quality and through our analysis, apply them in an aggressive but still controlled fashion. Using real database systems and workloads, we demonstrate that our system improves the state-of-the-art in terms of solution quality while being an order of magnitude faster.

# Bibliography

[1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for microsoft sql server 2005. In *VLDB*, pages 1110–1121, 2004.

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB 2000*.

[3] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the SIGMOD Conference*, New York, NY, USA, 2004. ACM Press.

[4] Vokan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez, Omar Ghattas, Eui Joong Kim, Julio Lopez, David O'Hallaron, Tiankai Tu, and John Urbanic. High resolution forward and inverse earthquake modeling on terascale computers. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 52, Washington, DC, USA, 2003.

[5] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The priority r-tree: a practically efficient and worst-case optimal r-tree. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 347–358. ACM Press, 2004.

[6] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, 1990.

[7] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact representations of separable graphs. In *SODA '03*, pages 679–688, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[8] Alexander Brodsky, Catherine Lassez, Jean-Louis Lassez, and Michael J. Maher. Separability of polyhedra for optimal filtering of spatial and constraint data. pages 54–65, 1995.

[9] Nicolas Bruno and Surajit Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *SIGMOD '05*.

[10] 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.

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

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

[13] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *SIGMOD '82: Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1982. ACM Press.

[14] 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.

[15] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL server. In *Proceedings of VLDB 1997*.

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

[17] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A formal perspective on the view selection problem. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 59–68, Rome, Italy, 2001. Morgan Kaufmann.

[18] Shirley Cohen, Patrick Hurley, Karl W. Schulz, William L. Barth, and Brad Benton. Scientific formats for object-relational database systems: a study of suitability and performance. *SIGMOD Rec.*, 35(2):10–15, 2006.

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

[20] Istvan Csabai, Marton Trencseni, Laszlo Dobos, Peter Jozsa, Geza Herczegh, Norbert Purger, Tamas Budavari, and Alexander S. Szalay. Spatial indexing of large multidimensional databases. In *CIDR*, 2007.

[21] Jochen Van den Bercken, Bernhard Seeger, and Peter Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings of VLDB '97*, pages 406–415, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[22] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. In *SCG '01: Proceedings of the seventeenth annual symposium on Computational geometry*, pages 106–114, New York, NY, USA, 2001. ACM Press.

[23] Luc Devroye, Christophe Lemaire, and Jean-Michel Moreau. Expected time analysis for delaunay point location. *Comput. Geom. Theory Appl.*, 29(2):61–89, 2004.

[24] Mike Folk, Quincey Koziol, and James Laird. Requirements for an indexing prototype in hdf5, 2005.

[25] Volker Gaede and Oliver Guenther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[26] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[27] Jim Gray. The revolution in database architecture. Technical Report MSR-TR-2004-31, Microsoft Research, 2004.

[28] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alexander S.Szalay, David DeWitt, and Gerd Heber. Data managemenet in the coming decade. Technical Report MSR-TR-2005-10, Microsoft Research, 2005.

[29] Jim Gray, Alexander S. Szalay, Gyorgy Fekeite, William O'Mullane, Maria A.Nieto Santisteban, Aniruddha R.Thakar, Gerd Heber, and Arnold H. Rots. There goes the neighborhood: Relational algebra for spatial data search. Technical Report MSR-TR-2004-32, Microsoft Research, 2004.

[30] Oliver Guenther. The design of the cell tree: An object-oriented index structure for geometric databases. In *Proceedings of ICDE'89*, pages 598–605, Washington, DC, USA, 1989.

[31] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57. ACM Press, 1984.

[32] G.Valentin, M.Zuliani, D.Zilio, and G.Lohman. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of ICDE 2000*.

[33] Marios Hadjieleftheriou. *www.cs.ucr.edu/ marioh/spatialindex*.

[34] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. pages 205–216, 1996.

[35] Gerd Heber and Jim Gray. Supporting finite element analysis with a relational database backend part i: There is life beyond files. Technical Report MSR-TR-2005-49, Microsoft Research, 2005.

[36] Gerd Heber and Jim Gray. Supporting finite element analysis with a relational database backend part iii: Where the numbers come alive. Technical Report MSR-TR-2005-151, Microsoft Research, 2005.

[37] Gerd Heber and Jim Gray. Supporting finite element analysis with a relational database backend part ii: Database design and access. Technical Report MSR-TR-2006-21, Microsoft Research, 2006.

[38] C. Heeren, H. V. Jagadish, and L. Pitt. Optimal indexing using near-minimal space. In *PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2003.

[39] H. V. Jagadish. Linear clustering of objects with multiple atributes. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 332–342. ACM Press, 1990.

[40] J.Gray, D.Slutz, A.Szalay, A.Thakar, J.vandenBerg, P.Kunszt, and C.Stoughton. Data mining the SDSS skyserver database. Technical Report MSR-TR-2002-01, Microsoft Research, 2002.

[41] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *Proceedings of the second international conference on Information and knowledge management*, pages 490–499. ACM Press, 1993.

[42] Howard Karloff and Milena Mihail. On the complexity of the view-selection problem. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 167–173, New York, NY, USA, 1999. ACM Press.

[43] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129, 1998.

[44] S. Leutenegger, J.M. Edgington, and M.A. Lopez. STR: A simple and efficient algorithm for R-Tree packing. In *Proceedings of ICDE 1997*.

[45] Swanwa Liao, Mario A. Lopez, and Scott T. Leutenegger. High dimensional similarity search with space filling curves. In *Proceedings of ICDE 2001*.

[46] Sam Lightstone and B. Bhattacharjee. Automating the design of multi-dimensional clustering tables in relational databases. In *Proceedings of VLDB '04*, 2004.

[47] Julio C. Lopez, David R. O'Hallaron, and Tiankai Tu. Big wins with small application-aware caches. In *Proceedings of Supercomputing '04*, page 20, Washington, DC, USA, 2004.

[48] Kwan-Liu Ma. Parallel volume ray-casting for unstructured-grid data on distributed memory architectures. In *Proceedings of PRS '95*, 1995.

[49] Tanu Malik, Randal C. Burns, and Amitabh Chaudhary. Bypass caching: Making scientific databases good network citizens. In *Proceedings of the ICDE Conference*, 2005.

[50] Bongki Moon, H. v. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE TKDE*, 13(1):124–141, 2001.

[51] Maria A. Nieto-Santisteban, Alexander S. Szalay, Aniruddha R. Thaka, William J. OMullane, Jim Gray, and James Annis. When database systems meet the grid. Technical Report MSR-TR-2004-81, Microsoft Research, 2005.

[52] William O'Mullane, Nolan Li, Maria A. Nieto-Santisteban, Alexander S. Szalay, Aniruddha R. Thakar, and Jim Gray. Batch is back: Casjobs, serving multi-tb data on the web. Technical Report MSR-TR-2005-19, Microsoft Research, 2005.

[53] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of SIGMOD '89*, pages 295–305, 1989.

[54] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS '84*, pages 181–190, New York, NY, USA, 1984. ACM Press.

[55] 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.

[56] R. Rew, G.P. Davis, S. Emmerson, and H. Davies. Netcdf user's guide for c. an interface for data access, 1997.

[57] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. In *SIGMOD '85*, pages 17–31, New York, NY, USA, 1985. ACM Press.

[58] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *SIGMOD 1979*.

[59] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *The VLDB Journal*, pages 507–518, 1987.

[60] Jonathan Richard Shewchuk. Tetrahedral mesh generation by delaunay refinement. In *Symposium on Computational Geometry*, pages 86–95, 1998.

[61] S.Papadomanolakis and A.Ailamaki. AutoPart: Automated schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, 2004.

[62] Alexander Szalay, Jim Gray, and Jan vandenBerg. Petabyte scale data mining: Dream or reality? Technical Report MSR-TR-2002-84, Microsoft Research, 2002.

[63] Alexander S. Szalay, Jim Gray, Ani R. Thakar, Peter Z. Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. The SDSS skyserver: public access to the sloan digital sky server data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 570–581. ACM Press, 2002.

[64] http://hdf.ncsa.uiuc.edu/hdf4.html. Hierarchical data format (HDF).

[65] http://my.unidata.ucar.edu/content/software/netcdf/docs/netcdf/Not-DBMS.html. Network common data form: Netcdf is not a database management system:.

[66] Office of Science Data Management Workshops. The office of science data-management challenge. Technical report, Department of Energy, 2005.

[67] Transaction Processing Performance Council (TPC). TPC benchmark H (decision support) standard specification.

[68] Tiankai Tu and David R. O'Hallaron. Extracting hexahedral mesh structures from balanced linear octrees. In *IMR '04, Proceedings of the 13th International Meshing Roundtable*, 2004.

[69] Tiankai Tu, David R. O'Hallaron, and Julio C. Lopez. Etree - a database-oriented method for generating large octree meshes. In *IMR'02, Proceedings of the 12th International Meshing Roundtable*, 2002.

[70] Tiankai Tu and David R.O'Hallaron. A computational database system for generating unstructured hexahedral meshes with billions of elements. In *Proceedings of the Supercomputing Conference*, 2004.

[71] Shyh-Kuang Ueng, Christopher Sikorski, and Kwan-Liu Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transctions VCG'96*, 3(4):370–380, 1997.

[72] Hongfeng Yu, Kwan-Liu Ma, and Joel Welling. A parallel visualization pipeline for terascale earthquake simulations. In *Proceedings of the Supercomputing Conference*, 2004.

[73] 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*, pages 1087–1097, 2004.