PARALLEL RELATIONAL OLAP

By

Todd Eavis

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
DALHOUSIE UNIVERSITY
HALIFAX, NOVA SCOTIA
JUNE, 2003

Your file Votre référence
ISBN: 0-612-83717-3
Our file Notre référence
ISBN: 0-612-83717-3

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "Parallelizing the Data Cube" by Todd Eavis in partial fulfillment for the degree of Doctor of Philosophy.

Dated: July 23, 2003

External Examiner:

Research Supervisor:

Examining Committee:

Departmental Representative:

DALHOUSIE UNIVERSITY

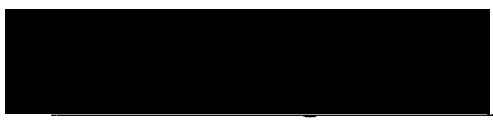
Date: **June 27, 2003**

Author: **Todd Eavis**

Title: **Paralellizing the Data Cube**

Department: **Computer Science**

Degree: **Ph.D.** Convocation: **October** Year: **2003**



Signature of Author

To the two women in my life: Amber and Bailey.

Table of Contents

List of Tables

List of Figures

Abstract

On-line Analytical Processing (OLAP) has become a fundamental component of contemporary decision support systems and represents a means by which knowledge workers can efficiently analyze vast amounts of organizational data. Within the OLAP context, one of the more interesting recent themes has been the computation and manipulation of the *data cube*, a relational model that can be used to represent summarized multi-dimensional views of massive data warehousing archives.

Over the past five or six years a number of efficient sequential algorithms for data cube construction have been presented. Given the size of the underlying data sets, however, it is perhaps surprising that relatively little effort has been expended on the design of load balanced, communication efficient algorithms for the parallelization of the data cube. This thesis investigates such opportunities, with a particular emphasis upon coarse-grained, distributed memory parallel architectures. New parallel algorithms for the computation of both the complete data cube and the partial data cube are presented. In addition, a model for distributed multi-dimensional indexing is proposed. The associated parallel query engine not only supports efficient range queries, but query resolution on non-materialized views and views containing hierarchical attributes. All of the proposed algorithms and data structures have been fully implemented and evaluated on contemporary distributed memory parallel machines.

List of Acronyms

API Application Programming Interface

APL A Programming Language

ASL Affinity Skiplist

AVL Adel'son-Velskii and Landis tree

BFS Breadth First Search

BPP Breadth First Writing, Partitioned Parallel BUC

BSP Bulk Synchronous Parallel

BUC Bottom Up Computation

ccNuma Cache Coherent NUMA

CGM Coarse Grained Multi-computer

CPU Central Processing Unit

CRCW Concurrent-read, Concurrent-write

CREW Concurrent-read, Exclusive-write

DBMS Database Management System

DMA Direct Memory Access

DSS Decision Support System

ERCW Exclusive-read, Concurrent-write

EREW Exclusive-read, Exclusive-write

FASMI Fast Analysis of Shared Multi-dimensional Information

GB Giga Byte

GHz Giga Hertz

GIS Geographic Information System

GM Glenn's Messages (Myrinet)

GNU GNU's Not Unix

HPCVL High Performance Computing Virtual Lab

I/O Input/Output

IT Information Technology

LAN Local Area Network

LEDA Library of Efficient Data Structures and Algorithms

logP Latency/Overhead/Gap/Processors

Mb Megabit

MBB Minimum Bounding Box

MCST Minimum Cost Spanning Tree

MFLOP Million Floating Point Operations per Second

MIMD Multiple Instruction Multiple Data

MISD Multiple Instruction Single Data

MMST Minimum Memory Spanning Tree

MOLAP Multi-dimensional OLAP

MPI Message Passing Interface

MPP Massively Parallel Processor

NAS NASA Advanced Supercomputing Division

NP Non-deterministically Polynomial

NUMA Non-uniform Memory Access

OLAP On-line Analytical Processing

OLTP On-line Transaction Processing

PC Personal Computer

PE Processing Element

PRAM Parallel Random Access Machine

PVFS Parallel Virtual File System

PVM Parallel Virtual Machine

RAM Random Access Memory

RCUBE Relational Cube

RDBMS Relational DBMS

ROLAP Relational OLAP

RP Replicated Parallel BUC

SIMD Single Instruction Multiple Data

SISD Single Instruction Single Data

SPMD Single Program Multiple Data

SQL Structured Query Language

STL Standard Template Library

STR Sort Tile Recursion

TB Tera Byte

TCP/IP Transmission Control Protocol/Internet Protocol

TPL Total Path Length

UDP User Datagram Protocol

UMA Uniform Memory Access

VIA Virtual Interface Architecture

Acknowledgements

Thanks to Dr. Gao, Dr. Milios, Dr. Cercone, and Dr. Bhavsar for your time and your patience.

Thanks Michelle for a fresh pair of eyes.

Thanks Andrew for your input and your commitment to getting this thing done.

And a giant thank-you to Amber for your indefatigable and often under appreciated support.

Chapter 1

Introduction

Effective data collection and management has always been a cornerstone of corporate success and changes in the economic landscape over the past decade have only served to intensify the associated computational requirements. The maturation of the global Internet and its graphical sibling, the World Wide Web, has lead to an exponential increase in the amount of data that corporations collect, manage, and analyze. And while the past two decades have also witnessed impressive increases in computing power, it is nonetheless true that large scale data warehousing and decision support systems now tax even the the most powerful uni-processor machines.

This thesis explores the use of parallel algorithms and data structures in the context of high performance On-line Analytical Processing (OLAP). OLAP is the foundation for a wide range of essential business applications, including sales and marketing analysis, planning, budgeting, performance measurement and data warehouse reporting [56, 88]. To support this functionality, OLAP relies heavily upon a data model known as the *data cube* [50, 57]. Conceptually, the data cube allows users to view organizational data from different perspectives and at a variety of summarization levels. In fact, the data cube model is central to our parallelization efforts. We note that this is a particularly rich area for new parallel research, one that builds on the significant amount of work that has already been performed in the sequential setting [57, 10, 50, 101, 105, 122, 3]. In particular, scalable methods are required

for the construction, querying, and analysis of OLAP data on contemporary parallel machines.

Our approach to the design of parallel OLAP systems has been to first identify sequential algorithms that have proven themselves to be both practical and efficient in an OLAP setting. By exploiting these optimized procedures on each of the compute nodes of a parallel machine, we have been able to focus our own attention more completely on the key parallel issues of load balancing and communication efficiency. The Coarse Grained Multicomputer or CGM model of parallel computing [33] shapes our algorithm design decisions and is used primarily because it is intended to capture the characteristics of current, practical computing architectures rather than purely analytical models such as the PRAM.

Many of the problems addressed in this thesis are either NP-complete or do not have tight bounds, even in the sequential setting, and we consequently explore a number of heuristic techniques. These heuristics are evaluated in the context of a three-phase evaluation strategy:

1. Use simulations to explore algorithmic trade-offs associated with load balancing and communication.

2. Design, implement, and optimize robust applications and then systematically evaluate them.

3. Use the feedback and insight obtained via the first two steps to tune the existing algorithms and to suggest new approaches to be explored.

While this thesis has a strong algorithmic component, and includes formal analysis where possible, the emphasis is ultimately upon *systems* and experimental research. The reason for this approach is worth noting. Asymptotic analysis of parallel algorithms is notoriously unreliable as a predictor of real performance on real machines.

Because of a plethora of commercial architectures — including propriety and sometimes exotic communication fabrics — the transition from "paper to silicon" is often a disappointing exercise. In the parallel setting there is simply no single reliable architectural abstraction that would be comparable to the von Neumann model representative of sequential computing architectures. As such, assumptions about issues such as latency, memory hierarchies, and computational bottlenecks are often not borne out in practice.

This thesis addresses an instance of *applied* computer science, namely data warehousing, and as such it is imperative not only that we build upon well motivated heuristics, but that we demonstrate clearly and convincingly that our techniques can and do work on existing parallel machines. Throughout the thesis, we attempt to strike a balance between a presentation of the high-level algorithmic work, detailed discussions of the implementation considerations, and experimental evaluations.

1.1 Overview of Primary Research

In this section we briefly discuss the key elements of this thesis. Though each element can be described and studied in isolation, we note that collectively they represent a cohesive approach to parallel OLAP.

1.1.1 Parallelizing the Data Cube

We have designed and implemented parallel algorithms for the efficient generation of the data cube. We work within the *relational* paradigm to produce a set S of 2^d summarized views (where d is the number of dimensions). Our general approach is to partition the workload in advance, thereby allowing the construction of individual views to be fully localized. Initially, we compute an optimized task graph, called a schedule tree, that represents the cheapest way to compute each view in the data cube from a previously computed view. We then employ a partitioning strategy based on

a modified *k-min-max* partitioning algorithm [6] that divides the schedule tree into p equally weighted sub-trees (where p is the number of processors). We support schedule tree construction with a rich cost model that accurately represents the relationship between the primary algorithmic components of the design. In addition, we present efficient algorithms and data structures for the physical disk-based materialization of the views contained in each sub-tree. Algorithm design choices are supported through both analysis and experimentation.

The end result of this process is a system that is load balanced, produces minimal communication (network utilization has been reduced to a single transfer of task information) and is computationally efficient. Extensive experimental evaluation has been conducted on a variety of data sets (both synthetic and "real world"), and we have examined performance for a range of values on d, p and n (record count). The results confirm parallel efficiency of 80% to 95% on processor counts from 1 to 24, and a near linear performance curve for increases in view count and data set size. We also note that in terms of "raw performance", a 24-node Linux cluster has been used to produce a 1.2 Terabyte, 14-dimensional data cube in just over one hour.

Preliminary reports of this research are described in [25, 26, 29, 27].

1.1.2 Computing Partial Cubes in Parallel

The complete data cube, as originally defined by Gray et. al. [50], consists of all 2^d possible views. When d and/or n, the size of the underlying data set, gets large, generating the entire data cube may be neither feasible nor desirable [101]. In Chapter 4, we describe a suite of new sequential algorithms for the generation of efficient *partial cube* schedule trees, where a partial cube represents a user-selected subset S of the 2^d views in the full space. In addition to the views initially selected for construction, we show that the inclusion of intermediate (i.e., non-selected) views can significantly reduce computation time. Since existing proposals for the partial cube problem,

such as a minimum Steiner tree approximation [105], are simply infeasible in high-dimensional space, we propose new methods based on a *greedy* approach. We extend our sequential methods to the parallel setting by passing the partial cube schedule tree to the partitioning algorithm described in Chapter 3 for parallel execution.

Experimental evaluation has shown that, for the special case of full cube construction, the new methods are capable of producing schedule trees whose weight is less than 1% greater than that of the best existing data cube scheduling algorithm. More importantly, for true partial cube construction, the new algorithms generate trees that are 25% to 70% cheaper than those produced by more naive approaches.

Preliminary results have been reported in [28, 30].

1.1.3 Parallel Multi-dimensional Indexing

Once the views of the data cube have been materialized, they are capable of supporting arbitrary queries on their summarized records. Such queries might be associated, for example, with user-directed visualization or sophisticated data mining operations. In either case, two of the most important forms of OLAP queries are the point query (a match against a single record) and the range query (arbitrary ranges on one or more of the d dimensions). Due to the size of the generated data cube views, it is generally not possible to effectively support point and range queries in a high dimensional OLAP context without some form of multi-dimensional index. However, for high dimension spaces, most conventional indexing methods perform quite poorly [9, 45]. Furthermore, the performance penalties associated with dynamic indexing techniques make most existing methods impractical in large data warehousing environments.

In Chapter 5 we propose a parallel indexing technique called the RCUBE that is optimized for the OLAP setting. It is based on a *packed R-tree* model [102], a design in which data is preprocessed so that efficient R-tree indexes can be generated. In using packed R-trees, we propose a sophisticated packing strategy based upon

the Hilbert space filling curve [70] that improves performance in high dimensional spaces. Since one of our primary objectives is to provide high performance OLAP indexing solutions, our R-trees are constructed as distributed data structures; queries are answered in parallel on each node that contains a portion of the relevant view. The proposed indexing model has been completely integrated into the data cube infrastructure. Users can query views within full or partial cubes, including ones that contain attribute hierarchies. Efficient index updates are also supported. In the case of partial data cubes, we introduce the concept of surrogate views and show how queries on views that have not been materialized can often be efficiently answered using these surrogates.

Experimental evaluation demonstrates that for arbitrary queries on a 16-processor parallel machine, the imbalance with respect to the number of records retrieved per node is less than 0.3%, while the corresponding speedup for parallel query resolution is close to linear. In high volume environments, the parallel query engine can resolve 1000 random multi-dimensional range queries on a 20 GB data cube in less than 10 seconds. Moreover, post-processing overhead associated with view surrogates and attribute hierarchies has been shown to add just 5% to 15% to the time taken for query resolution.

Preliminary results have been reported in [32, 31].

1.2 Our Parallel Design Model

Given the strong systems orientation of this thesis, the hardware, software, and design models associated with *parallel* computation are of particular importance. Our goal has been to avoid those abstractions and models that are not predictive of observed performance when real systems are constructed. Instead, we have tried to integrate a deep understanding of contemporary parallel computing issues into the algorithm design process. For an introduction to the concepts and terminology relevant to such

an approach, see Appendix A.

In terms of the current research, we note that MIMD computer systems [43] have been used throughout the thesis to implement parallel algorithms and data structures that are relevant to high performance data warehousing. Two distinct physical architectures have been used for performance evaluation purposes. The first parallel machine — and our primary implementation platform — is a fully distributed Linux cluster [95]. The cluster is an example of a "commodity" system — it consists of open source software and low cost, widely available hardware.

Primary software components include:

- NPACI Rocks (RedHat based Linux)
- RedHat GNU C/C++/Fortan Compiler version 2.96
- GNU C/C++/Fortran Compiler version 3.2.2
- LAM (Local Area MultiComputer)/MPI version 6.5.6
- Portable Batch System (OpenPBS version 2.2p5)
- MAUI scheduler version 3.0.5.p11
- PVM 3.4.3
- standard development tools like make, emacs, vi, automake, autoconf, etc.

Primary hardware components include:

- 32 nodes (dual processors), with each node contained in a 1U 19" case
- 64 1.8 GHz Intel Xeon (x86) processors (2 per node)
- 32 GB distributed memory (1 GB per node shared between 2 processors)
- 2.56 TB of distributed external memory (two 40 GB IDE hard disks per node)

- 100 MBit/s 100Base-Tx (Fast Ethernet) switched interconnect
- front-end node is connected to a UPS

A second machine, a more traditional 24-CPU SunFire multi-processor, located at the High Performance Computing Virtual Lab [62] was used for final testing on the data cube algorithms discussed in Chapter 3.

SunFire hardware includes:

- 24 x 900 Mhz (8 MB E-Cache) UltraSPARC-III processors
- 24 GB of memory shared in a cc-NUMA design
- Gigabit Ethernet Network Interface Cards
- Multiple Fibre Channel Arbitrated Loop (FC-AL) host adapters
- 11.7 TB of Sun StorEdge T3 Fibre Channel disk technology (disk array)

SunFire software includes:

- Soloris 8 Operating System
- Sun Forte Developer 6 (C, C++, Fortran compilers and debuggers)
- Sun HPC ClusterTools 4.0
- Sun Grid Engine 5.3 Enterprise Edition

We note that the SunFire was used because it is configured with a *disk array.* A disk array is a coordinated collection of I/O devices that is seen by the operating system (and, by extension, user processes) as a single logical device. On-board hardware stripes data (by byte or by block) across the disk units so that multi-process reads and writes on a single data store can be processed in parallel. The justification for using a disk array will be presented in Chapter 3.

In all cases, the Message Passing Interface [83] was used for the communication subsystem, providing us with a number of significant benefits. First, the very rich API of MPI allowed us to improve performance by exploiting a number of its less trivial primitives. Second, and perhaps more importantly, the broad acceptance of the MPI standard guarantees a form of portability that is not possible with any other system. Not only does it allow us to port the code from our Linux testbed to virtually any other distributed memory platform with a minimum of effort, but it also allows us to move the code to SMP-based machines [24] since these platforms almost always have efficient MPI implementations. This is an advantage worth noting since many distributed memory algorithms work quite well in shared memory environments (the actual coding may be more difficult of course), while a movement in the opposite direction — from OpenMP [89] to MPI — is much more problematic.

Algorithmically, our focus has been the coarse-grained computing paradigm synonymous with MIMD architectures. To avoid the performance penalties often associated with message passing on most MIMD machines (particularly the cluster), algorithms have been constructed so as to reduce the quantity of message transfers to a minimum while, at the same time, exploiting the power of today's microprocessors. This is a style of design directly supported by the CGM model [33] (though BSP [121, 12] and logP [23] analysis could also be used to justify our design choices). Standard performance metrics such as Parallel Speedup and Efficiency are used to support algorithm and data structure design decisions. The end result is a software infrastructure that, in BSP terminology, efficiently bridges the gap between abstract algorithm design and real architectures and applications.

1.3 A Look Ahead

The thesis is organized as follows. Chapter 2 provides an overview of Online Analytical Processing, including such things as fundamental OLAP operations and server

architectures. The chapter also presents the data cube operator and describes a number of the algorithms that have been designed for its efficient construction.

The succeeding chapters present the core contributions of this thesis, including the proposed algorithms, the implementation issues, and the experimental results. Chapter 3 focuses on the construction of the complete data cube and describes the workload partitioning algorithm, the costing model, and the algorithms and data structures for view materialization. The related problem of partial data cube construction is discussed in Chapter 4. A new greedy model is presented, including a suite of algorithms for the generation of efficient partial cube schedule trees in high dimensional space. Chapter 5 then presents our new parallel RCUBE model. We discuss methods for partitioning, building, and updating the supporting multi-dimensional indexes. As well, we present the details of a parallel query engine, one that includes OLAP-specific functionality for resolving queries on partial cubes and views containing attribute hierarchies. Finally, in Chapter 6, we briefly describe possible future work.

Chapter 2

An Introduction to OLAP and the Data Cube

2.1 Introduction

While database and data management systems have always played a vital role in the growth and success of corporate organizations, changes to the economy over the past decade have seemed to exaggerate their significance. Both communication patterns and corporate/client relationships have evolved within the context of a new digital age. To keep pace, IT departments have begun to exploit rich new tools and paradigms for processing the wealth of data and information generated on their behalf. Along with relational databases, the venerable cornerstone of corporate data management, knowledge workers and business strategists now look to advanced analytical tools in the hope of obtaining a competitive edge.

In this chapter, we examine the current trends, technologies, and terminology relevant to an understanding of *On-line Analytical Processing* or OLAP [17, 19, 34], perhaps the most ubiquitous of today's sophisticated processing models. We provide an introduction to the general area of *decision support systems* (DSS) in Section 2.2, making a distinction between OLAP and two other primary DSS components, *information processing* and *data mining*. Section 2.3 defines OLAP in terms of its core operations and functionality. Section 2.4 discusses the data warehouse, the

fundamental structural building block for virtually all OLAP systems. We present the *data cube* model in Section 2.5, and discuss a number of important sequential algorithms for data cube construction in Section 2.6. Section 2.7 concludes the chapter with a brief summary.

2.2 Decision Support Systems

As the name implies, decision support systems are designed to empower the user with the ability to make effective decisions regarding both the current and future state of an organization. To do so, the DSS must not only encapsulate static information, but it must also allow for the extraction of patterns and trends that would not be immediately obvious. Users must be able to visualize the relationships between such things as customers, vendors, products, inventory, geography, and sales. Moreover, they must understand these relationships in a chronological context, since it is the time element that ultimately gives meaning to the observations that are formed.

Discussions of DSS applications often implicitly assume that such systems are part and parcel of a single homogeneous technology. In fact this is typically not the case. Rather, knowledge-based systems come in a number of distinct forms, each extending the functionality of the core Database Management System (DBMS) [56, 17, 34, 75]. Below, we describe the three primary DSS models, including OLAP which is the focus of our current research.

- **Information Processing.** Here, we are concerned with fundamental querying and reporting functions. IT professionals typically design queries — whether "ad hoc" or pre-defined — that extract detailed information directly from the supporting database management systems. Some form of visualization module may also be utilized to streamline the user interface. Analysis at this point is likely to be quite simple, consisting of sorting and basic aggregation, and lacks the sophistication to uncover anything but the most obvious features of the

stored data.

- **OLAP**. Online Analytical Processing extends the basic reporting capabilities of Information Processing systems by allowing a robust multidimensional analysis of the archived data from a variety of perspectives and hierarchies. Operations such as *drill-down*, *roll-up* and *pivot* (to be discussed in Section 2.3) provide insights into corporate growth, spending, and sales patterns that would simply not be possible otherwise. Additional OLAP functionality may include operations for ranking, moving averages, growth rates, statistical analysis, and "what if" scenarios. Note that the terms "DSS" and "OLAP" are sometimes used interchangeably. However, in this thesis OLAP refers to only one approach to the design of DSS functionality.

- **Data Mining**. This third class represents the "natural evolution" of knowledge-based data management systems. In this case, our goal is to automate the discovery process so that trends and patterns can be retrieved with minimal user input. The patterns are not necessarily those that are embedded directly in the aggregated fields of the data warehouse. Rather, they may consist of subtle regularities that cross hierarchical and/or dimension boundaries and, as such, would be less likely to be discovered by conventional OLAP techniques. Typical data mining operations include *classification* (categorization of novel items), *association* (identification of patterns and trends), and *cluster analysis* (identification of data groupings via multi-attribute similarity).

2.2.1 The Historical Context of OLAP

The multi-dimensional analysis of data can in some sense be traced back to the introduction of the programming language APL in the early 1960s [66]. While particularly obscure syntactically, APL was interesting from a data analysis perspective in that

it explicitly supported the use of multi-dimensional variables. Nevertheless, its "unfriendly" nature prevented it from being more widely used.

A decade or so later, the first true multi-dimensional product — Express [37] — arrived on the market. It stored data in an array-based format and permitted some degree of dimensional analysis. Over the next ten or fifteen years, numerous other decision support products, from companies such as Comshare [20], Metaphor, and Pilot [96], were developed. Over time, these systems moved away from time-shared mainframe implementations towards client/server network-centric applications. Still, none were widely supported since they either required excessive hardware resources, were limited in functionality, or were non-intuitive to use. In fact, during this time period, it was the *spreadsheet* that was most often associated with multi-dimensional analysis.

By the 1990s, processing power had grown to such a degree that serious data-intensive applications were now completely viable on cost-effective PC networks. In addition, the emergence of the Internet/Web lead to a marked increase in the amount of digital information available for analysis. The combination of these two factors eventually encouraged the introduction of a whole new generation of advanced multi-dimensional tools, a trend that has continued into the present time frame. In fact, current projections set the value of the OLAP market at five billion dollars by 2004 [88]. Figure 2.1 depicts annual growth rates for successive years, beginning in 1994.

At present, the list of major "players" includes companies such as Hyperion, Cognos, Microsoft, Oracle, MicroStrategy and Business Objects. Microsoft, in particular, with the introduction of its low cost SQL Server 7.x and Analysis Services 2000, has become the fastest growing vendor and will likely challenge Hyperion as the market leader within the next two or three years. Conversely, many of the original powerhouses, such as Pilot, Gentia, and Informix, have succumbed to competitive pressure and have all but vanished.

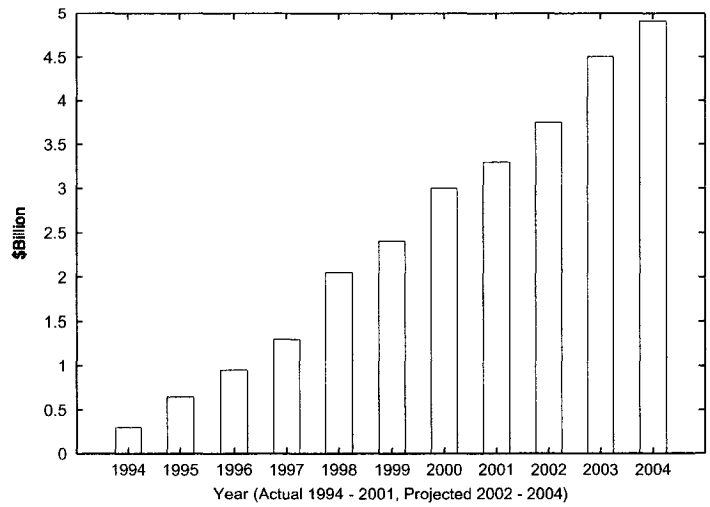


Figure 2.1: Worldwide total OLAP market share in billions of dollars.

In summary, the message to be drawn from this short introduction is that, after a lengthy evolutionary period, OLAP has emerged as a pervasive and crucial element of decision support systems. At present, the list of vendors includes some of the world's largest software providers. Given the significance of the area, it is perhaps not surprising that OLAP-centric publications have become increasingly common in the database literature during the past five or six years.

2.3 Defining OLAP

Despite the long history of applications for multi-dimensional analysis, the term "OLAP" was not coined until 1992. In that year, E. F. Codd, the originator of the relational database model, produced a report entitled "Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate" [19] that formally identified the new field. The following list, taken from that report, identifies the twelve features that would/should make up any OLAP application:

1. **Multidimensional conceptual view.** This supports "slice-and-dice" operations and is usually required in financial modelling.

2. **Transparency**. OLAP systems should be part of an open system that supports heterogeneous data sources. Furthermore, the end user should not have to be concerned about the details of data access or conversions.

3. **Accessibility**. OLAP should present the user with a single logical schema of the data.

4. **Consistent reporting performance**. Performance should not degrade as the number of dimensions in the model increases.

5. **Client/server architecture**. Requirement for open, modular systems.

6. **Generic dimensionality**. Not limited to 3-D and not biased toward any particular dimension. A function applied to one dimension should also be able to be applied to another.

7. **Dynamic sparse-matrix handling**. Related both to the idea of nulls in relational databases and to the notion of compressing large files, a sparse matrix is one in which not every cell contains data. OLAP systems should accommodate varying storage and data-handling options.

8. **Multiuser support**. OLAP systems need to support multiple concurrent users, including their individual views or slices of a common database.

9. **Unrestricted cross-dimensional operations**. Similar to Rule 6; all dimensions are created equal, and operations across data dimensions do not restrict relationships between cells.

10. **Intuitive data manipulation**. Ideally, users shouldn't have to use menus or perform complex multiple-step operations when an intuitive drag-and-drop action will do.

11. **Flexible reporting**. Save a tree. Users should be able to print just what they need, and any changes to the underlying financial model should be automatically reflected in reports.

12. **Unlimited dimensional and aggregation levels**. A serious tool should support at least 15, and preferably 20, dimensions.

The list is largely self-explanatory and clearly emphasizes the multi-dimensionality of the data and the ease with which users should be able to access it. However, while significant in that it was the first meaningful attempt to describe the OLAP environment in a structured manner, it is worth noting that the report did not become the definitive industry standard — as had Codd's earlier work on relational databases [18]. Perhaps some of the public skepticism stems from the fact that the report was commissioned by Arbor Software, a leader in the OLAP application field. Nevertheless, it remains one of the few formal OLAP specifications.

2.3.1 OLAP: A Functional Definition

Though the Codd report indirectly emphasized the functionality that a commercial OLAP application should possess, it did not explicitly define the core OLAP operations. In fact, there are five such fundamental features that have come to be synonymous with the OLAP label. The list below presents these functions, while graphical examples are provided in Figures 2.2, 2.3, and 2.4.

- **Roll-up**. The roll-up operation collapses the hierarchy along a particular dimension(s) so as to present the remaining dimensions at a coarser level of aggregation. Figure 2.2 illustrates how the "location" dimension, originally listed in a city-by-city fashion, is aggregated in order to provide provincial totals.

- **Drill-down**. In contrast, the drill-down function allows users to obtain a more detailed view of a given dimension. Again, in Figure 2.2, we see how the "product" dimension is broken down from its initial, broad categories into product-specific listings.

- **Slice**. Here, the objective is to extract a slice of the original cube corresponding to a single value of a given dimension. No aggregation is required with this operation. Instead, we are allowing the user to focus in on values of interest. Figure 2.3 illustrates the process for a single value of the "product dimension".

- **Dice**. A related operation is the *dice*. In this case, we are defining a subcube of the original space. In other words, by specifying value ranges on one or more dimensions, the user can highlight meaningful blocks of aggregated data. In Figure 2.3, a subset of dimension values on product, location, and customer have produced the 3 x 2 x 2 subcube.

- **Pivot**. The pivot is a simple operation that allows OLAP users to visualize cube values in more natural or intuitive ways. While Figure 2.4 provides a simple example with a symmetrical $4 \times 4 \times 4$ cube, the pivot operation can be equally effective with dimensions of varying cardinality.

2.3.2 OLAP: The FASMI Definition

The OLAP Report — an independent journal that provides product and marketing information for the OLAP industry — has itself defined a metric they call FASMI or *Fast Analysis of Shared Multidimensional Information* [88]. The FASMI definition succinctly describes the criteria that may be used to grade or compare OLAP products and, though simple and decidedly informal, it has nevertheless become one of the most commonly referenced OLAP metrics. Its criteria are presented in the following list:

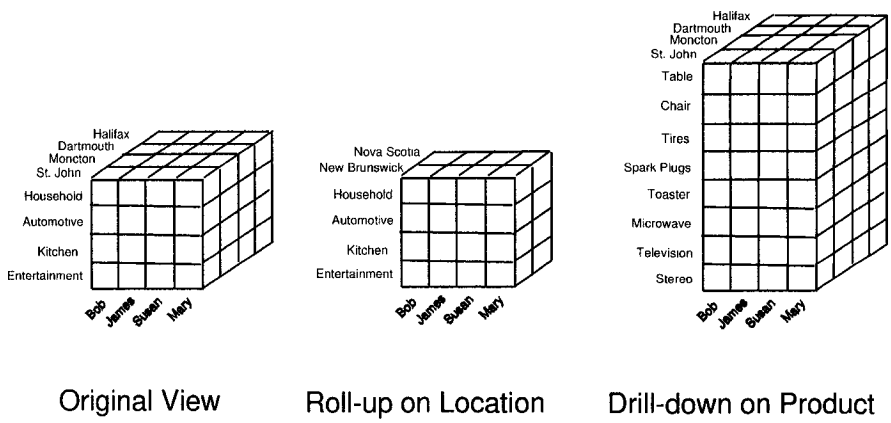


Figure 2.2: Roll-up and Drill-down on a simple three-dimensional cube.

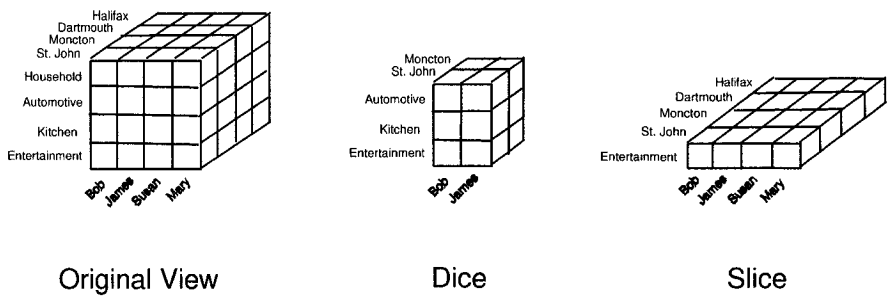


Figure 2.3: Slicing and Dicing a three-dimensional cube.

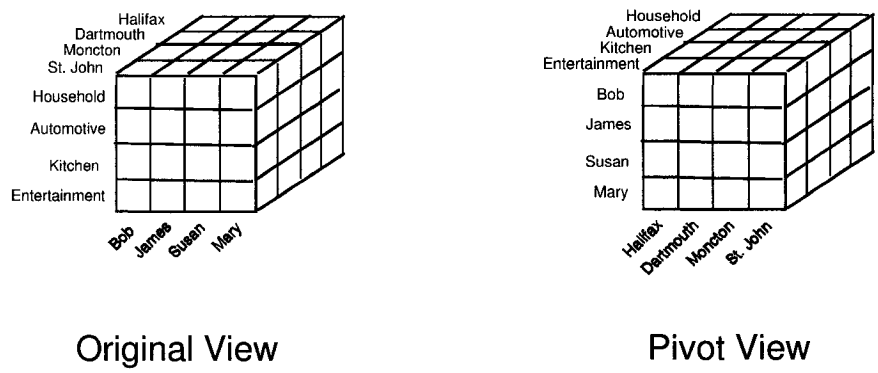


Figure 2.4: The Pivot operation.

- **Fast**. Vendors must be able to efficiently trade off pre-calculation costs and storage requirements with real-time query response. Studies have shown that users are likely to abort queries that take longer than thirty seconds to complete.

- **Analysis**. Tools should not only provide the five fundamental operations but extras such as times series analysis, currency translation, and data mining capabilities. Most of the business and analytical logic should be available without sophisticated 4GL programming.

- **Shared**. Security and concurrency control should be available when required. As stated earlier, however, most OLAP systems assume that user-level updates will not be necessary.

- **Multidimensional**. This is the *key* FASMI requirement. Regardless of the characteristics of the physical implementation, the user must see the data in subject-oriented hierarchies.

- **Information**. Applications must be able to handle vast amounts of data. Again, regardless of the server model that is used, good OLAP applications may have to support data cubes that scale to the terabyte range.

The FASMI list is significant in that it reduces the more technical Codd report to a concise subset of intuitive requirements. More to the point, it highlights three of the primary objectives of the current research — building an OLAP computational model that is fast, inherently multi-dimensional, and scalable to very large data warehouses.

2.4 The Data Warehouse

While OLAP can be defined in terms of the functionality it offers, it is important to note that sophisticated contemporary OLAP systems are almost always constructed on top of a physical data management system known as a *data warehouse*. In fact,

our own research can be described as providing parallel OLAP processing in the context of a data warehousing environment. It is therefore important to understand the philosophy and design of these systems.

The classic definition of the data warehouse was provided by W. H. Inmon when he described it as a "subject-oriented, integrated, time-variant, and non-volatile collection of data in support of management's decision-making process" [65]. What does this imply? And why can we not simply use our production databases to support OLAP processing and data mining activities?

To answer the second question first, there are a number of reasons why day-to-day operational databases, often referred to as On-line Transaction Processing or OLTP systems, are particularly ill-suited to decision support. The most salient of these issues are as follows:

- OLTP databases contain detailed, transaction-oriented records. While this degree of detail may be appropriate for day-to-day processing activities, it is not well-suited to the problems of classification and trend analysis.

- Production systems record current information, typically by day or by week. Again, this is inappropriate for decision support since one of the key components of such systems is the ability to provide a complete history of the activity of the organization.

- For the most part, OLTP systems are *customer-oriented*, in that they focus on transaction activities that are driven by individual purchases (or some similar metric). For decision support systems, on the other hand, we require a perspective that helps to identify *all* market forces.

- Because operational databases function in high volume, read/write environments, they require sophisticated mechanisms for record locking and data recovery. Conversely, apart from periodic updates, decision support systems should

only require read access to the data. As such, the server can be constructed so as to provide more efficient access characteristics.

For supporting this kind of OLAP functionality, data warehouses provide an effective alternative to the transaction-oriented environment of the operational database. They are organized around subjects, rather than atomic transactions. They represent aggregated or summarized information from a variety of sources. They house data collected over very long periods, typically years. And they are tuned for read-only access. In other words they represent a "subject-oriented, integrated, time-variant, and non-volatile collection of data in support of management's decision-making process."

2.4.1 Architecture

In general, data warehouses can be seen as three-tiered data models [56, 17]. Figure 2.5 provides a graphical representation. Information is first extracted from operational sources and then cleaned, transformed and loaded into the data warehouse. We will refer to this pre-processing stage as Level 0. Though this first step is itself outside the scope of the data warehouse proper, it is nonetheless a crucial activity that has received considerable attention from OLAP vendors. Often, the production data resides in a collection of remote, heterogeneous repositories and must undergo considerable *massaging* before it can be integrated into a single clean store.

In any case, once the data has been culled from the remote sources, it is placed into the data warehouse at Level 1, which at this point in time is almost always a relational DB. The data warehouse itself may be constructed as a monolithic enterprise-wide entity and/or a series of *data marts*, each containing some subset of the corporate data. In either case, it will be the job of the OLAP server at Level 2 to actually supply analytical functionality for the DSS system. In practice, there are two forms of OLAP servers, known as ROLAP and MOLAP (discussed in greater detail in Section 2.4.3), that may be used for this purpose. We note that while their conceptual aims are quite

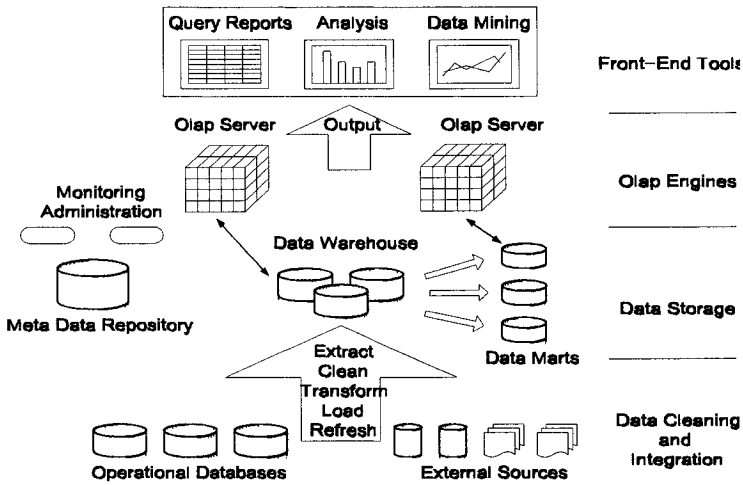


Figure 2.5: The three-tiered OLAP model.

similar, their internal data representations are quite different. Finally, in the top-tier, we find the front end tools that provide a user-friendly (often graphical) interface for the knowledge workers who will exploit the system.

2.4.2 The Star Schema

Operational databases are typically designed using the *Entity-Relationship* model. Here, objects or entities of interest are defined, along with the sometimes numerous and complex relationships between them. This type of database is well-tuned for transaction processing which typically requires rapid insertion/deletion of small records. However, it is not particularly well-suited for data warehousing, where the goals are (i) resolving complex queries and (ii) batch loading and updating. In particular, we do not want numerous join operations on many small tables; such processing would be far too expensive for the multi-dimensional queries typical of OLAP computing.

Instead, the standard database design for OLAP is what's known as the *Star Schema*. Figure 2.6 provides a simple illustration. In this particular case, the Star

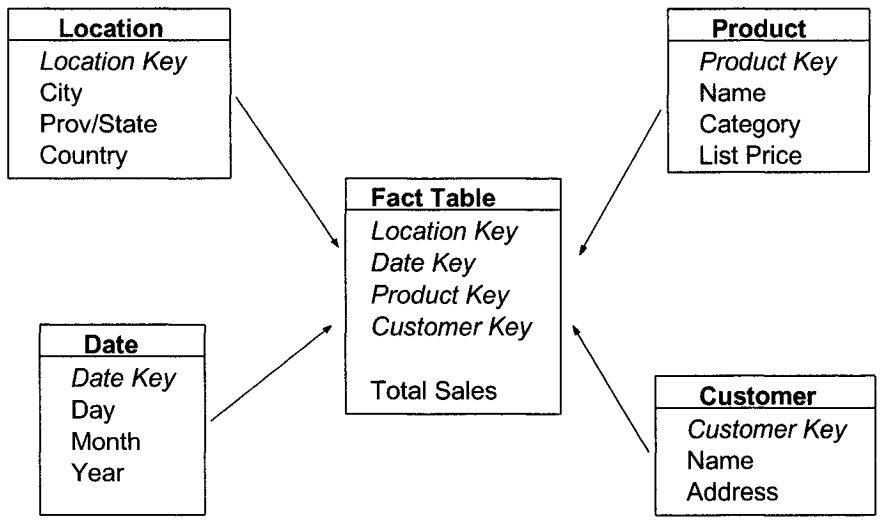


Figure 2.6: A four-dimensional Star Schema.

Schema identifies a group of four dimensions that are of interest to the user group. Each of these dimensions is associated with a table or relation that encapsulates the core information necessary for a complete understanding of that dimension. One of the fields in the dimension table serves as the *primary index* (depicted in italics).

In addition to the dimension tables, a *fact table* is also included. The fact table houses the *detail* records for the data warehouse and consists of the dimensional coordinates (primary keys from the dimension tables), plus the value of one or more numeric aggregations. In this case, we are describing "Total Sales" as it relates to each unique combination of dimensional values. For example, a particular record in the fact table might provide us with the value of all purchases of boating equipment at the Halifax location for Customer John Smith during March of 2002.

This simple arrangement is appropriate for data warehousing because it allows rapid querying/loading of the main fact table and because it reduces join operations by creating a minimal number of secondary dimension tables. Moreover, it maps very nicely to the multi-dimensional perspective of the data that end users will actually visualize.

As a final note, we add that more complex design frameworks are possible. Such extensions might be required, for example, when dimensions can be sub-divided into *hierarchies* of sub-attributes (a full discussion of hierarchies will be provided in Chapter 5). Nevertheless, even in such cases, the Star Schema still serves as the fundamental design model.

2.4.3 MOLAP, ROLAP and Multi-dimensional Data

One of the most important elements of OLAP environments is their reliance upon multi-dimensional data values. As mentioned previously, data warehouses represent subject-oriented records rather than transaction-oriented ones. As such, aggregated values can be viewed as existing within a logical *cube*, where the user is free to index the cube on one or more dimensional axes. In the nomenclature of OLAP, this type of conceptual representation of the data gives rise to what is known as a *data cube* (see Section 2.5 and Figure 2.8). It is this cube-like model, in fact, that is the focus of the second tier of the DSS architecture, the OLAP server.

Since the data cube suggests a multi-dimensional interpretation of the data space, a number of OLAP vendors have chosen to physically model the cube as a multi-dimensional array. These multi-dimensional OLAP (MOLAP) products offer rapid response time on OLAP queries since, in theory at least, it is possible to index directly into the data cube structure to retrieve subsets of aggregated data. Unfortunately, MOLAP solutions have not proven to scale effectively to large, high-dimensionality data sets [10] (though MOLAP capacity has grown significantly in recent years). The problem is that as the number of dimensions grows, the data in the data cube becomes increasingly *sparse*. In other words, many of the attribute combinations represented by the data cube structure do not contain any aggregated data. As such, a fully materialized MOLAP array can contain an enormous number of empty cells, resulting in unacceptable storage requirements [101]. Though compression techniques

are often used to alleviate this problem, doing so destroys the natural indexing that makes MOLAP so appealing. As a result, awkward hybrid indexing schemes — that combine both sparse and dense sub-arrays — are required.

In contrast, relational OLAP (ROLAP) seeks to exploit the maturity and power of the relational paradigm. Instead of a multi-dimensional array, the ROLAP data cube is implemented as a collection of relational tables, each representing a particular view of the data. Because the views are now conventional database tables, they can be processed and queried with traditional RDBMS techniques (e.g., indexes and joins). More importantly, however, they can be more efficient on very large data warehouses since only those data cube cells that actually contain information are housed within the tables. The "empty" portions of the space do not have to be represented in any way. On the downside, there is no "built-in" indexing with a ROLAP cube as there would be with a MOLAP implementation. Instead, all attribute values within the record must be included with the aggregated or summary values so that the record's position within the cube can be determined. One might liken this to a *fully-qualified path name* in operating system terminology. This additional overhead tends to offset some of the space savings, particularly in dense environments. Furthermore, the absence of an implicit index implies that an explicit one must be provided. In practice, this can be a challenge of considerable importance since efficient multi-dimensional indexing techniques are notoriously complex.

2.5 The Data Cube

In the previous section, we informally introduced a multi-dimensional structure called the data cube, describing it as a data abstraction that allows one to view aggregated data from a number of perspectives. In this section, we will formalize the data cube concept and introduce a number of important sequential algorithms for its computation that have been reported in the literature.

Before proceeding, however, we introduce some of the basic terminology necessary for a thorough understanding of the problem. We begin by noting that a standard OLAP analysis environment consists of a group of dimensions, each of which has been identified by the data warehouse designers as being of interest to the user community. Dimensions are also known as *attributes*; we will use these terms interchangeably throughout the remainder of the thesis. Attributes can be of two types. *Feature* attributes refer to those dimensions that represent entities or concepts central to the structure of the organization. Examples would be things such as customer and product. *Measure* attributes, on the other hand, refer to the items of interest, the values that will be aggregated in terms of the feature attributes. We note that while there may be many feature attributes, in most cases there will be a very small number of measure attributes (often just one). Our example in Section 2.4.2 identified "Total Sales" as the measure attribute; an aggregate sales figure was associated with each unique combination of feature attributes in the fact table.

Measure attributes may be calculated using functions from one of three distinct categories.

- **Distributive**. Functions in this category have the unique feature that when computed across independent data partitions, the partial results can be combined into a single aggregate. Examples are *sum*, *min*, and *max*.

- **Algebraic**. Simply put, an algebraic function is one that can be produced by combining distributive functions. Examples would include *average* and *standard deviation*.

- **Holistic**. Functions in this category cannot be decomposed into algebraic functions. *Median* and *rank* are common examples. In general, holistic functions are difficult to compute efficiently and practical implementations often resort to approximations.

For convenience, we will utilize a single distributive measure attribute throughout the thesis, namely *summation*. Virtually all data cube related research papers employ this same simplification.

We also note that with a d-dimension space, each of the d attributes $\{A_1, A_2 \ldots A_d\}$ has a *cardinality* that identifies the number of unique values for that attribute. For example, if one of the data cube dimensions is "Product", and there are 275 individual products in our database, then the cardinality of Product, denoted $|Product|$, is 275. We refer to the group of d cardinalities as the *cardinality set* C.

In total, a d-dimensional data warehouse is associated with 2^d views (sometimes referred to as the view *Power Set*). In OLAP terminology, views are also known as *cuboids* or *group-bys*. Again, we will use these terms interchangeably throughout the remainder of the thesis. Each view or cuboid represents a distinct combination of feature attributes, and can be seen as depicting an aggregation of the measure attribute at a given level of *granularity*. For example, given the attribute set ABC, we say that the aggregated view A is of *coarser* granularity than the aggregated view AB. Note that we are interested in attribute *combinations*, not *permutations*, since the order of the attributes does not matter. In practice, we usually substitute letter labels for the attribute names. For example, Customer may be attribute 'A', Product may be attribute 'B', etc. The "Customer/Product" cuboid is therefore simply referred to as AB.

The relationship between the 2^d views — in terms of common attributes — is typically represented by a *lattice* [57]. See Figure 2.7 for a graphical illustration. Starting with the *base cuboid* — the finest granularity view containing the full complement of d dimensions — the lattice branches out by connecting every parent node with the set of child nodes/views that can be derived from its dimension list. In other words, the attributes of a parent view must be a *superset* of the attributes of a child view. A parent containing k dimensions can be connected to k views at the next level in

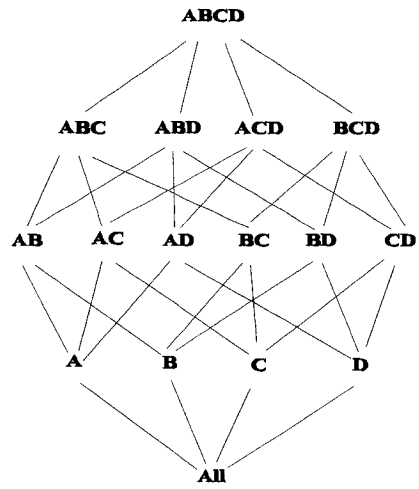


Figure 2.7: The data cube lattice consists of all possible attribute combinations. The "all" node represents the aggregation of all records.

the lattice, each of which contains $k - 1$ attributes. Conversely, a child view can be associated with $d - k$ parents (if this is not obvious, note that because the lattice is perfectly symmetrical, the number of parents for a given view at level k is equivalent to the number of children for a given view at level $d - k$). Finally, it should be understood that parent/child relationships are not *exclusive* — parents can share common children just as children may have common parents.

Conceptually, then, the data cube consists of the base cuboid, surrounded by a collection of $2^d - 1$ sub-cubes/cuboids that represent the aggregation of the base cuboid along one or more dimensions. Since the base cuboid contains *all* feature attributes, it can be used to compute all of the other coarser cuboids by aggregating across one or more of its component dimensions. In other words, it may be possible to initially compute only a subset of all possible views, leaving the materialization of the remaining views to some later time (if necessary). As such, a data cube can be described as *full* if it contains all 2^d possible views, or *partial* if only a subset of views has actually been constructed.

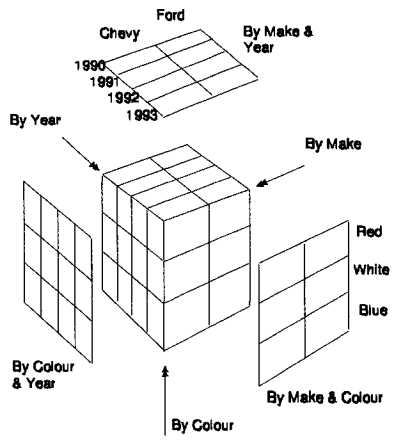


Figure 2.8: A three dimensional data cube.

Figure 2.8 depicts a small, practical data cube example from the automotive industry. Note that this is actually a partial cube since the single dimension views are not depicted. This particular data cube has three feature attributes — make, colour, and year — and a single measure attribute — sales. Sales is computed with the distributive *sum* function. By selecting cells (a "point" query), planes (a "slice" query), or sub-cubes (a "dice" query) from the base cuboid, we can analyze sales figures at varying granularities.

One final note is in order at this point. We have described the data cube as a conceptual model. However, in the case of a MOLAP server, it is also the physical model, as MOLAP stores the cube structure directly as a (possibly compressed) multi-dimensional array — typically drawn from the Star Schema described in Section 2.4.2.

2.5.1 The Data Cube Operator

Strictly speaking, no special operators or SQL extensions are required to take a raw data set, composed of detailed transaction-level records, and turn it into a data structure, or group of structures, capable of supporting subject-oriented analysis. Rather,

the SQL *group-by* and *union* operators can be used in conjunction with 2^d sorts of the raw data set to produce all cuboids. However, such an approach would be both tedious to program and immensely inefficient, given the obvious inter-relationships between the various views. Consequently, in 1995, the data cube *operator*— an SQL syntactical extension — was proposed by Gray et al. [50] as a means of simplifying the process of data cube construction.

2.6 Data Cube Algorithms

Subsequent to the publication of the seminal data cube paper, a number of independent research projects began to focus on designing efficient algorithms for the computation of the *full* data cube [3, 10, 57, 101, 105, 122]. Most were based upon the exploitation of the data cube lattice. It should be clear from the lattice depiction that many views share common dimension values and that any efficient computational mechanism for producing group-bys must exploit these relationships. For example, a three-dimensional cuboid — say ABC — can be viewed as the parent of three two-dimensional cuboids — AB, AC, BC — each of which contains a distinct combination of two dimensions of the parent. Clearly, it should not be necessary to independently compute all four views since the parent and one or more of the children may be able to share some portion of the aggregation workload.

For convenience, we may group the primary data cube algorithms into three general categories: *top down*, *bottom up*, and *array-based*. In this section, we will look at an example of each. We note that it is important to understand the mechanics of the sequential algorithms — at least from a conceptual standpoint — since parallel coarse grained algorithms often exploit the strengths of existing single processor techniques.

2.6.1 Top Down

The top down methods work directly from the lattice to compute smaller group-bys from larger parents. For example, the parent view ABCD might be used to generate ABC, AB and A. Perhaps the most well known technique from this class is the PipeSort [105]. In fact, the PipeSort forms the basis of our parallel data cube algorithms in Chapter 3 and Chapter 4. For this reason, we will describe the algorithm in some detail.

Conceptually, the PipeSort works from fine granularity views to coarse granularity views. It is a "relational" or ROLAP algorithm in the sense that is works directly from, and with, standard relational tables. The designers of the algorithm identified a number of goals or features that should be exploited when determining how to most efficiently construct a particular view. That list includes:

- **Smallest Parent**. As the name would suggest, this optimization tries to compute a view from the smallest previously computed parent.
- **Cache Results**. If views are small enough to fit into memory, we should use them to compute child views before writing the original view to disk.
- **Amortize Scans**. Since more than one child contains a subset of the attributes of a given parent, we should use that parent to compute multiple child views whenever possible.
- **Share Sorts**. If a sort-based technique is employed, a single sort should be shared by output views containing common attributes.

It should be noted that some of these optimization objectives may be contradictory. The goal, then, for an algorithmic implementation is to find effective trade-offs that are robust across variations in data set size and dimension count. In the case of the PipeSort, the objective is to extract an appropriate *minimum cost spanning*

tree (MCST) from the original lattice. The PipeSort MCST represents a connected graph of 2^d nodes and $2^d - 1$ edges, with each vertex having an *in-degree* of exactly one. In other words, the MCST represents a unique "plan" in which the the cost of traversing edges — and thereby building cuboids — will be minimized.

The PipeSort algorithm is presented in Algorithm 1. The algorithm works by setting the sort orders for the group-bys at successive levels, starting from the bottom of the tree and working upwards. We note that although the tree is constructed in a "bottom-up" fashion, the physical views of the data cube itself will be constructed top down from the complete MCST. Hence the top-down classification. At each level, we effectively decide the most efficient way to define the sort orders of level $k + 1$, given the orderings that have been defined for level k in the previous iteration of the algorithm (Note that level 0 represents the bottom of the lattice). We note that for a given group-by in level $k + 1$, only one child view in level k can share the same sort order; the parent view must be completely re-sorted in order to construct any other children. The most "efficient" mapping from level $k + 1$ to level k must therefore take into account the costs associated with re-sorting a given view versus simply scanning an already sorted view. We refer to these two cost measures as the *Sort* cost and the *Scan* cost, respectively.

Algorithm 1 Sequential PipeSort

Input: A lattice of 2^d nodes augmented with "sort" and "scan" costs.
Output: A minimum cost spanning tree.

```
1: for level k = 0 to level d − 1 do
2:    Generate-Plan(k + 1 → k)
3:    for each group-by g in level k + 1 do
4:       Fix the sort order of g as per the order of the group-by in level k that is
         connected to g by a "scan" edge
5:    end for
6: end for
```

It should be clear that the key step in Algorithm 1 is the call to the *Generate-Plan* function. Algorithm 2 explains how Generate-Plan actually works. Any two levels k and $k+1$ can be represented as a *bipartite graph*. Note that a bipartite graph $G(V, E)$ is one in which the vertices $v \in V$ can be divided into two disjoint sets $\{V_c, V_r\}$ such that there is no edge $e(v_1, v_2) \in E$ with endpoints $(v_1, v_2) \in V_c$ or $(v_1, v_2) \in V_r$. In other words, edges never connect vertices in the same partition. Given the bipartite representation, we may reduce the plan generation to a *weighted bipartite matching* problem. We note that a *matching* on a bipartite graph defines a subset of edges $M \subseteq E$ such that for all vertices $v \in \{V_c, V_r\}$ at most one edge is *incident* on v, where "e incident on v" implies $e(v, u)$ or $e(u, v)$. Simply put, no two edges in M share the same endpoint. A *maximum weighted bipartite* matching, then, finds a matching on $G(V, E)$ such that when a numeric weight $w(e)$ is associated with each edge, the combined aggregate weight $\sum_{e \in M} w(e)$ is maximized.

The Generate-Plan function uses the weighted bipartite matching algorithm to define the *minimum* cost mapping from level $k + 1$ to level k. We note that by "inverting" the edge weights — $w(e) = max(w) - w(e)$, where $max(w)$ is the heaviest edge in E — a maximum matching becomes a minimum matching. In order to use the weighted matching algorithm, we augment the bipartite graph by adding k copies of each group-by in level $k + 1$ and associating these copies with the sort cost of the group-by. The original group-by is given the scan cost. Once augmented, the weighted bipartite matching can be performed, leaving a minimum cost edge set connecting the two levels. The final step is simply to re-order the attributes of the group-bys in level $k + 1$ to match those already defined on level k. Figure 2.9 provides a graphical illustration of the Generate-Plan algorithm.

When the PipeSort Algorithm has completed, we are left with a minimum cost spanning tree in which all sort and scan edges have been defined. Figure 2.10 provides a simple graphical illustration for a four-dimensional data cube. Note that a series of

Algorithm 2 Generate Plan

Input: The nodes of level k and $k + 1$.
Output: A minimum cost matching.
1: **for** level $k = 0$ to level $d - 1$ **do**
2: Create k additional copies of each level $k + 1$ group-by
3: Connect each new vertex to the same set of child vertices as the original
4: Assign "Sort" costs to the new edges and "Scan" costs to the original.
5: Find the minimum cost matching on the augmented bipartite graph.
6: **end for**

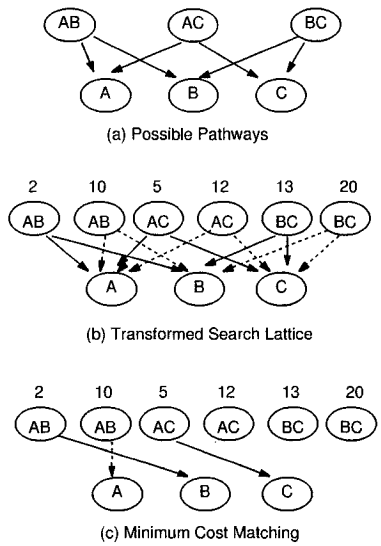


Figure 2.9: The parent level is augmented to include both sort and scan costs. Bipartite matching gives us the "cheapest" way of producing level k from level $k + 1$. Note: "scan" edges are dashed, while "sort" edges are solid.

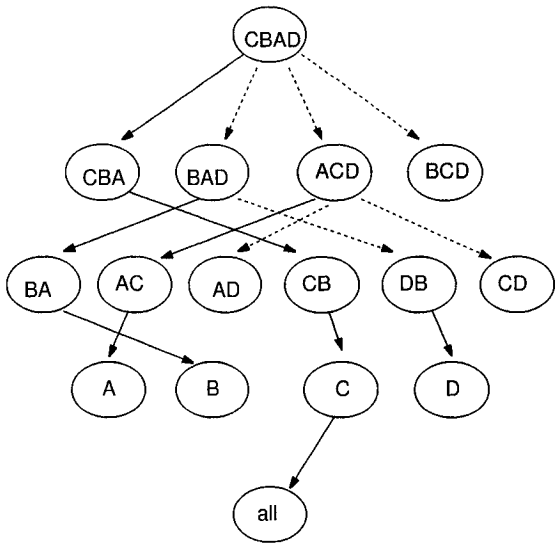


Figure 2.10: A four dimensional minimum cost PipeSort spanning tree.

prefix-ordered pathways or *pipelines* has been identified. As such, it is now possible to generate all the views in a pipeline — say CBAD → CBA → CB → C → all — with a single sort and scan. To do so we sort the input set (i.e., fact table) in the order CBAD. Now we make a linear pass through the sorted set, combining the *measure* value of common records into a single composite value. We note that all common records are contiguous since the set is sorted. Moreover, because all views in the pipeline share the same prefix-order, we may concurrently compute the aggregations on each of the views. For example, if a pair of contiguous record r_1 and r_2 have different feature attributes, then we can write the aggregate value for r_1 to the output file and initialize the new aggregate with the measure value of r_2. At the same time we can check each of the coarser granularity aggregates in the pipeline to see if an aggregate needs to be written to their output views. In fact, only one aggregation variable need be maintained for each group-by. Once the pass is complete, all views in the pipeline have been generated. The processing of all pipelines results in the generation of the *full* data cube.

2.6.2 Bottom Up

For each extra dimension that we add to the base cuboid, the total number of views in the fully materialized data cube exactly doubles. Moreover, as the dimensions increase, the high-dimension cuboids become increasingly sparse. In other words, many of the cells in the cube have no values in them. Sparsity ultimately results in a large number of views that are almost as big as the base cuboid. Since top-down algorithms tend to utilize views in the upper portion of the lattice as pipeline input sets (i.e, the parent of some set of views sharing a common prefix), sort costs can grow significantly in large sparse spaces.

To help reduce the penalty associated with the sorting of many large views, *bottom up* methods have been proposed. In this section, we describe one of the most well known — the *BUC* or Bottom-Up Computation algorithm [10]. Bottom up algorithms work by first aggregating (usually with a sort) on a single dimension, then recursively partitioning the current result set in order to aggregate at successively finer degrees of granularity. Algorithm 3 describes how this works in the case of BUC. The recursive algorithm takes as input a relation (i.e, a set of multi-dimensional records), plus the current dimension. Initially, the input relation is the fact table, while the current dimension parameter is the first attribute in the complete dimension list. We will describe how the dimensions are ordered below. In the first step, the input set is scanned and aggregated into a single output record. Step 2 will be explained shortly — it will be skipped it for now. In the main algorithm loop, we begin by determining the cardinality c of the current input set, where the cardinality represents the number of distinct values for the current dimension to be partitioned. Once we determine the cardinality, then we arrange the input set into c partitions. BUC uses sorting for this purpose. When the partitions have been defined, we iterate through the partitions, *recursively* calling the BUC algorithm, this time using the current partition as input, along with the incremented current dimension variable (so that we examine only

the remaining dimensions). When we have finished the current set of partitions, the algorithm will *backtrack* and process the next partition at the previous level of recursion.

Algorithm 3 The BUC Algorithm

Input: The partition to be aggregated, plus the current dimension d
Output: A single record that represents the aggregated input.

```
1: Aggregate input relation
2: if input count == 1 then
3:    Write ancestor records and return
4: end if
5: Write output record from Step 1
   {/* process remaining partitions of finer granularity */}
6: numDims = total number of dimensions in data cube
7: for dim = d; dim < numDims; dim++ do
8:    c = cardinality of dimension dim
9:    Partition input on its c unique values
10:   for each of the c partitions in the input set do
11:      Recursively call BUC(partition, dim + 1) using the current partition as input
12:   end for
13: end for
```

Figure 2.11 provides an illustration of the recursive subdivsion of a four dimensional space. In this case, BUC will first partition (i.e., sort) the input set on A. It will then aggregate on $<a1>$ before partitioning $<a1>$ into its $<a1b1>$ and $<a1b2>$ components. This recursive partitioning will continue until the last dimension has been reached. Eventually, the backtracking will return the algorithm to the $<a2>$ partition, at which point the whole process is repeated.

The BUC algorithm is well-suited to sparse, high dimension data cube problems for two main reasons.

1. In Step 2 of the algorithm, we check to see if the size of the current partition is equal to one. If it is, then we know that there is no value in continuing the recursion since no further partitioning can be performed. We therefore write

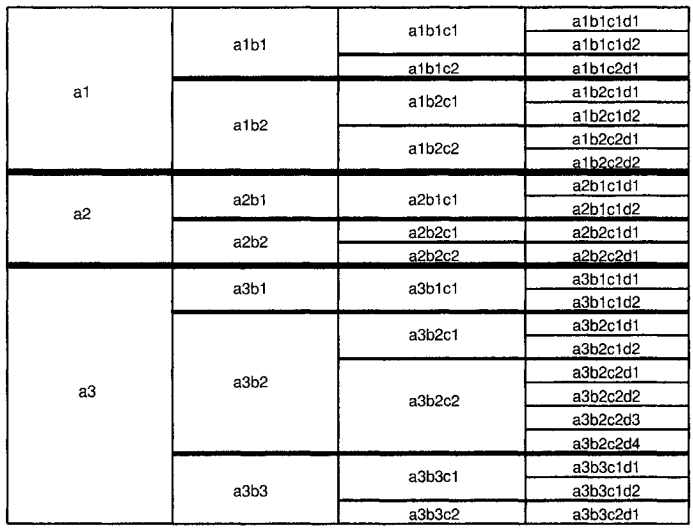


Figure 2.11: The bottom up "perspective". Partitioning proceeds left to right.

out the aggregates for all ancestors and return immediately. For example, when we encounter the tuple $<a1b1c2>$, we know that we can write the aggregate value for $<a1b1c2d>$ without further processing. Because many partitions will in fact have a size of one in sparse spaces, this *short circuiting* can significantly improve performance.

2. As it recursively partitions the input set, BUC divides the data into smaller and smaller segments. Consequently, it is increasingly likely that these partitions fit entirely into main memory, possibly reducing the reliance on more expensive external memory sorting.

As noted above, the order in which the attributes are partitioned is also important. Specifically, BUC partitions the attributes in order of decreasing cardinality. In so doing, it minimizes the use of large sorts since the maximum cardinality dimension will immediately split the data into as many small partitions as possible.

Experimental results for BUC, reported in [10], demonstrate approximately a factor of two performance advantage with respect to alternative data cube algorithms in sparse spaces. We note, however, that the benefit of BUC tends to be limited to these

full cube, high dimension problems. When denser views — the kind typically found in the lower levels of the lattice — are required, the authors themselves acknowledge that the time to recursively partition input sets can dominate the run time since very little short-circuiting takes place. As a result, BUC is ill-suited to problems in which large sparse views either do no exist or are not required.

2.6.3 Array-based

While the PipeSort and BUC algorithms work specifically with relational tables, another approach is to exploit data sets that have been encoded directly in an array-based format. This corresponds to the MOLAP model described in Section 2.4.3. Though much of the existing MOLAP work is associated with commercial products, and is not well described in the literature, Zhao et al. [122] have published a well-referenced paper that uses array-based computation. We will refer to their method as the ArrayCube algorithm.

ArrayCube works directly upon what are known as *chunked* arrays [106]. By chunking, we mean that instead of arranging a multi-dimensional array in conventional *row major* order (e.g, an $A \times B$ array would be written as A consecutive vectors of size $|B|$), the array is written to disk in block-size units, where each block constitutes a multi-dimensional sub-cube extracted from the parent array. Figure 2.12 provides a graphical example on a $16 \times 16 \times 16$ three-dimensional cube, in which each sub-cube is made up of a $4 \times 4 \times 4$ sub-array. In this case, the 64 chunks are stored on disk in the order indicated by the numbering pattern. With chunking, access patterns — both during cube computation and at query time — are significantly improved since one dimension is not "favored" over another. Figure 2.13 illustrates how chunking "equalizes" retrieval patterns, ensuring that in this case no more than two blocks are retrieved regardless of the dimension queried.

Unlike the ROLAP algorithms which generally rely upon sorting to identify records

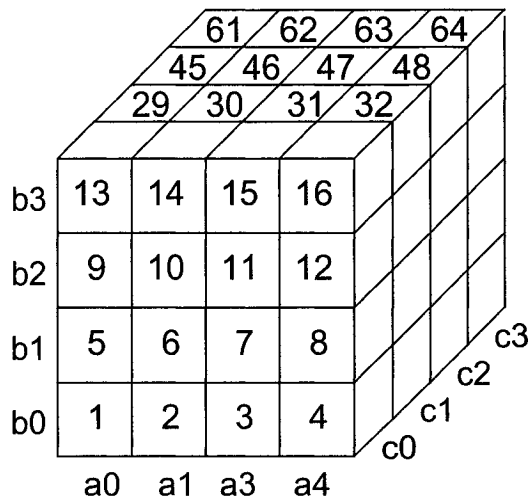


Figure 2.12: A three-dimensional chunked cube.

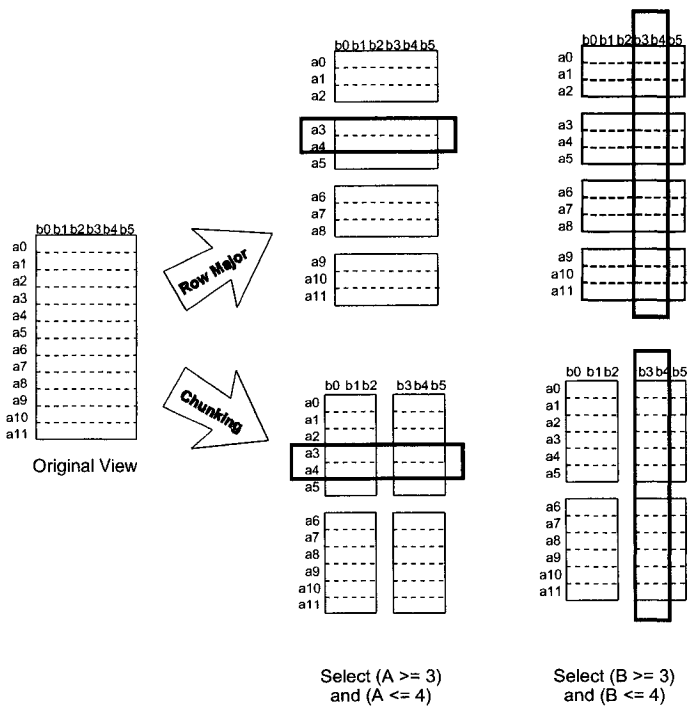


Figure 2.13: Access patterns for chunking versus row major storage.

with common attribute values, the ArrayCube uses the structure of the cube itself to support aggregation operations. Specifically, it "walks" through the array, chunk by chunk, aggregating values into cuboids at various granularities. No sorting is required since the indices of the array implicity define the values of the feature attributes. The chief contributions of the ArrayCube algorithm are :

1. It minimizes the memory required to materialize the various cuboids.

2. It identifies the parent cuboids that should be used to most efficiently compute child views.

The authors in [122] noted that for a given chunk order, some cuboids/sub-arrays could be constructed without materializing their entire contents in memory. In other words, these cuboids could be aggregated in sections. Once a given section or *partition* was completed, it could be immediately written to disk and its memory could be used for the next partition. They further observed that total memory requirements were minimized when the traversal pattern was ordered so that attributes were arranged in order of ascending cardinality. For example, if for a three-dimensional cube the attribute cardinality was given as $A = 10$, $B = 50$, $C = 100$, then the traversal order would be $A \rightarrow B \rightarrow C$. This is in fact the traversal order for Figure 2.12 — we walk along dimension A, then dimension B, then dimension C. Figure 2.14 illustrates how this is done for the AB, AC, and BC cuboids generated from the ABC base cuboid. As the picture suggests, BC can be computed with just one buffer since as we walk the cube in the given order, we can aggregate and write out results after every four chunks; no subsequent chunks need to be considered. This is not the case with the other two cuboids — we need to keep more buffers in memory. In fact, the AB cuboid requires the entire plane to be available.

In addition to the ordering issue, the ArrayCube authors proved that given multiple possible parents views, a child cuboid should always be computed from the parent

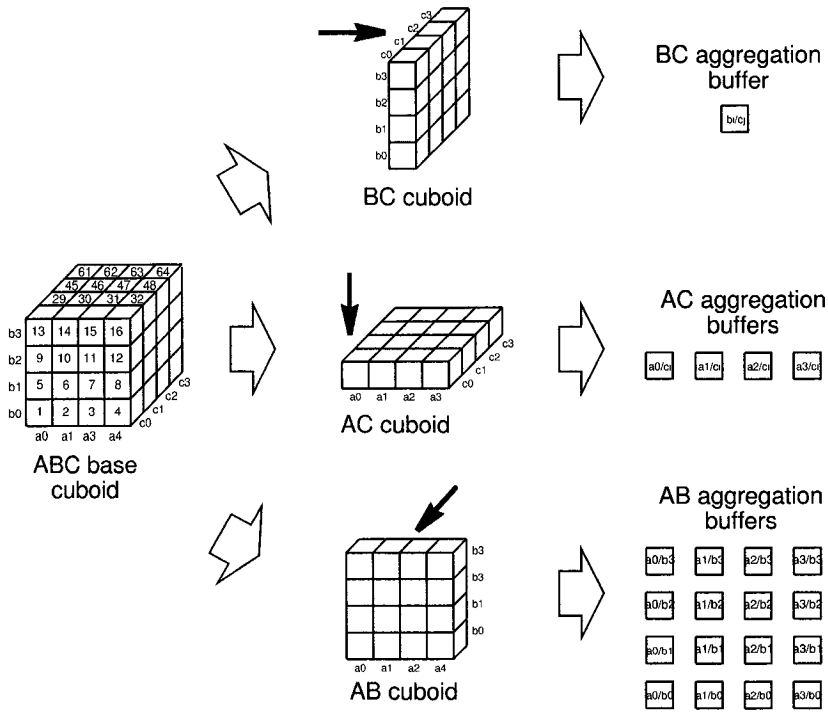


Figure 2.14: Generating group-bys with ArrayCube.

that housed the smallest possible prefix of the child. This is important since we do not want to compute each cuboid from all of the cells in the base cuboid — it is much more efficient to use smaller intermediate parent views, as is done in PipeSort. As a concrete example, given an $A \rightarrow B \rightarrow C$ traversal order, the cuboid B could be more efficiently computed (in terms of required memory) from AB than from BC since $\underline{B}C$ represents a prefix of size one on B, while AB contains no prefix whatsoever.

The algorithm utilizes these various observations to construct a *minimum memory spanning tree*. This MMST defines the ordering of the attributes, as well as the parent views used to compute each child. The complete process is described in Algorithm 4.

Note the recursive nature of Step 6. Once a cuboid partition has been completely processed, it is used to compute child cuboids while its cells are still in memory. In a conceptual sense, the ArrayCube can be seen as being composed of a series of pipelines, in much the same way as the ROLAP PipeSort uses sorting pipelines.

Algorithm 4 The ArrayCube Algorithm

Input: A d-dimensional data set in chunked format.
Output: A complete data cube in chunked format.

```
1: Arrange the dimensions by increasing cardinality.
2: Construct a MMST, obeying (i) the defined ordering and (ii) the minimal prefix
   rule
3: for each child view k , starting at the base cuboid do
4:    Concurrently aggregate cells into the cuboid buffer(s) for each child k
5:    if the current buffer for any k is fully aggregated then
6:       Recursively process the children of k
7:    end if
8:    Write the k partition to disk and make its memory available if necessary
9: end for
```

On dense data sets, there is little question that the ArrayCube is a very efficient algorithm. By avoiding sorting operations and thereby reducing data cube computation to a linear traversal of a chunked array, the ArrayCube provides impressive experimental numbers [122]. The drawback, however, is that even with the MMST, memory requirements for the algorithm can be excessive as the number of dimensions, and their associated cardinalities, grows. And while the algorithm can be re-organized as a "multi-pass" procedure to try to deal with this problem, it remains *memory sensitive*. Having said that, given a small dense data set, the algorithm can be effective. In fact, for data sets of no more than 500,000 records and five dimensions, the authors demonstrated that they could take *relational* data, convert it to chunked format, run the ArrayCube on it, and convert the data cube results back to relational storage faster than one of the existing ROLAP algorithms could accomplish the same task.

2.7 Conclusions

Decision Support Systems have become a prominent component of most corporate IT strategies. At the heart of the DSS sits the powerful combination of data warehousing and OLAP. In recent years, a significant amount of research has focused upon the exploitation of algorithms and data structures required to support large scale OLAP

functionality. While the industry continues to change — and grow — at a relatively dramatic pace, it is also true that there are enormous possibilities that remain to be explored in this area.

In this chapter, we have examined the concepts and functionality at the heart of On-line Analytical Processing and the data cube. We began with a discussion of the general area of Decision Support Systems, contrasting the sometimes overlapping fields of Information Processing, true OLAP, and Data Mining. Core OLAP operations were illustrated, with a particular emphasis on the multi-dimensional nature of the data and its associated processing. Finally, MOLAP and OLAP servers were examined in the context of the three-tiered DSS.

The data cube, both as a logical and physical construct, was also discussed in some detail. A number of optimized algorithms for data cube generation currently exist. The techniques were categorized and an example of each class was presented. The PipeSort, perhaps the most well-known of the top down techniques, forms the basis of our work on parallel data cube computation and will be further discussed in Chapter 3 and Chapter 4.

Chapter 3

Computing Full Data Cubes in Parallel

3.1 Introduction

One of most important recent research problems in the area of Decision Support Systems is the efficient implementation of the data cube [3, 10, 50, 57, 101, 105, 122]. In the sequential setting, a significant amount of data cube related work has already been carried out. As discussed in Chapter 2, the primary focus of that research has been upon algorithms (i) that reduce computation by sharing sort costs [3, 105], (ii) that minimize external memory sorting by partitioning the data into memory-size segments [10, 101], and (iii) that represent the views themselves as multi-dimensional arrays [50, 122]. By contrast, relatively little research effort has been focused upon parallel computation. The small amount of work that has been performed to date has shown itself to be either overly-simplistic or inefficient on existing parallel machines [48, 54, 84, 86].

In this chapter, we describe in detail the design and evaluation of a load-balanced and communication efficient parallel algorithm for full data cube construction. Our approach is to distribute view subsets to individual nodes, where efficient sequential algorithms can be used to independently calculate their assigned workload. Designed and optimized for distributed memory parallel machines, the new parallel algorithm

is suitable for multi-computers with and without a shared disk array.

We also present experimental results that demonstrate the viability of our approach. Our evaluation explores the impact of parameters such as the number of processors, the size of the input set, and the total number of views. In short, we demonstrate that our algorithm produces running times that are near optimal with respect to those of the underlying sequential approach. We also show that the algorithm scales effectively to larger problems. In fact, in recent testing using a 24-processor Linux cluster, data cubes exceeding one terabyte in size have been constructed in just over one hour.

The chapter is organized as follows. Section 3.2 reviews previous research in the area of parallel data cube construction. In Section 3.3, we present the motivation for our own work in the field. A detailed description of the new parallel algorithm is provided in Section 3.4, with Section 3.5 discussing algorithmic and system issues relevant to high performance. We present the supporting costing model in Section 3.6. In Section 3.7, we briefly describe the primary features of the physical implementation. More formal algorithm analysis is then provided in Section 3.8. Our experimental results are presented in Section 3.9. Section 3.10 is a review the chapter's objectives, with final conclusion provided in Section 3.11.

3.2 Related Work

Though a significant amount of data cube research has been performed in the sequential setting, parallel efforts in the area have been much less common, and arguably less successful. In this section we review the four most significant methods previously presented in the literature, and identify the benefits provided by these approaches as well as those features that might in fact lead to sub-optimal or irregular performance in practical environments.

Some of the most significant work on parallelizing the data cube that has been reported in the literature has been undertaken by Goil and Choudhary [48, 47, 49]. Here, the authors specifically target the MOLAP environment, constructing and processing array-based structures rather than relational tables. In terms of their approach to parallelizing the data cube workload, they have opted for a *data partitioning* model. Fundamentally, there are two primary means of parallelizing the data cube problem. One can either (a) *localize* view computation so that clusters of individual cuboids are constructed on each node or (b) fully distribute each view so that every processor computes a portion of every group-by. Many researchers are attracted to the second approach because of the *apparent* simplicity of equitably partitioning array-based structures across nodes. The Goil and Choudhary work takes this approach.

In [48, 47, 49], partitioning is performed by globally sorting the data set on a given dimension A such that, for a p processor parallel machine, the original data set is split into partitions A_1, A_2 ... A_p, one per processor. Furthermore, for $1 \leq i < j \leq p$, the partitioning guarantees that the value of A in any tuple of the locally sorted partition A_i is less than or equal to the value of A in any tuple of the local partition A_j. We note that there *may* be a single value of A that straddles partitions A_i and A_j, when $j = i+1$. It is important to note that when data partitioning is performed, partial results computed on distributed views may eventually have to be merged with the partial results on other nodes. For example, if the data is partitioned across processors on the attribute A, then all cuboids containing A as their *first* dimension (view orderings can be altered to accommodate this requirement) can be computed almost independently since there is at most one set of contiguous tuples with the same value of A that can be found on different processors. For cuboids not containing the attribute A, however, a merge of the partial results must be done. The standard approach for reducing merge costs is to globally sort/partition the base "A" view (i.e., the highest dimensional view containing the attribute A) on another attribute,

say B, at which point all those remaining views with B as the first attribute can be computed independently. The process is repeated d times for each distinct dimension. While this technique can significantly reduce re-partitioning costs, the total amount of network traffic due to redistribution can still be quite large.

As noted, this parallel design uses array-based structures for cube computation. For large problems — the ones ideally suited to a parallel machine — the authors recognized that main memory would simply not be large enough to concurrently house all of the necessary arrays. This is true even for parallel machines in that the addition of p processors or memory banks provides only a constant increase in memory capacity, while an increase in the number of dimensions causes the cardinality product $\prod_{i=1}^{d} C_i$ to explode in size. As a result, arrays must be carefully partitioned and controlled, a task that is supported by a powerful but complex memory manager. In effect, the memory manager serves as a virtual memory sub-system, swapping array segments to disk when necessary. For high cardinality spaces, however, even disk devices are insufficient for materialization of all 2^d cuboids when those cuboids are stored in an array format. Consequently, complex chunk compression techniques are required, with the result being that the data structures actually stored on disk look very little like traditional arrays.

The build algorithm itself attempts to utilize available memory to construct partitioned cuboids. Because of the irregular structure of the compressed arrays, the minimum memory spanning tree of the ArrayCube — and the simple chunk traversal pattern associated with it — cannot be used. Instead, the authors use a modified form of the PipeSort minimum cost spanning tree, employing bipartite matching to determine appropriate parent/child pairings. In this case, costing decisions must take into account the size of computed views (a difficult task given the unpredictable form of compressed chunks), network transfer speed (for re-partitioning distributed cuboids), and chuck I/O costs. Once data cube build decisions have been made the chunks

must be read into memory and aggregated. Since the chunks themselves may be compressed, additional overhead is required to process their contents. In some sense, this is akin to tuple hashing, a process shown in [105, 86] to be more expensive than sorting. Finally, we note that data skew and cluster patterns may result in uneven construction costs for cuboids of equivalent size. The end result is either sub-optimal load balancing or further redistribution costs.

The Goil and Choudhary research represents an impressive engineering effort, one that includes a number of interesting algorithmic observations. However, given the complexity of their approach and the apparent amount of overhead introduced, it is unclear how effective their method is likely to be in terms of parallel speedup. In their experimental results, the authors provide numbers for a group of larger processor counts (8 to 64) on an IBM SP2, but fail to list any figures for one to seven processors. This is an unfortunate omission in the parallel context in that it makes it impossible to associate speedup or efficiency measures with their work. Given the considerable processing overhead described above, this is a critical issue since an obvious concern with the implementation is whether all this overhead limits performance improvements when moving from a sequential environment. In other words, arbitrary run-time figures for eight nodes mean very little if these results are only marginally faster than a competitive sequential algorithm on one node. The absence of this type of comparison leaves considerable doubt as to the viability of their approach in practical environments. It also serves as a reminder that the simplest conceptual model — in this case, partitioned multi-dimensional arrays — is not necessarily the cleanest design *in practice.*

In [54], Lu, Huang, and Li describe a parallel data cube implementation for the Fujitsu AP3000, a high-end multi-computer. While this work is in the relational environment, it uses hashing for the aggregation of common records, rather than the sorting model discussed in the previous chapter. Here, the fields of each record are

concatenated to form a *hash key* that, in turn, identifies a unique *aggregation bucket.* If the record is associated with a bucket that has not yet been accessed, then the current record is added to the table. Otherwise, its value is added to the current total for that field combination. Of course, if collisions occur (i.e., two or more hash keys pointing to the same bucket), some form of collision resolution must be employed. Hashing for data cube computation was in fact first proposed in conjunction with the PipeHash [105], a sequential algorithm developed by the designers of the PipeSort. The technique is attractive because it is relatively simple to implement — as opposed to the somewhat more complicated sort/aggregation pipelines of the PipeSort — and because the $O(n)$ asymptotic upper bound for hash table construction would appear to provide a performance advantage over the sort-based methods that typically rely on $\theta(n \log n)$ sorting algorithms. However, experimental results in [105] clearly demonstrate the performance superiority of the PipeSort. The reason for this somewhat counter-intuitive result is that hashing costs cannot be shared amongst child group-bys since the field combinations for different views are completely unique, with the result that 2^d hash tables must be constructed for the generation of the full cube. Moreover, the "constants" involved in hashing with such large keys are significant. These two factors result in slower than expected computation time for hashed-based cubing.

With respect to the algorithm itself, parallelism was achieved by either (i) having processors share the costs of producing individual, partitioned hash tables or (ii) computing groups of hash tables on individual nodes (the paper's algorithm description is quite vague). Though they recognized that cuboids could be most efficiently computed from the smallest available parent in the lattice, the authors of the algorithm provided no means by which to achieve this cost reduction. Instead, a single common parent was used to produce all hash tables during a given iteration, where the group of available cuboids was drawn from a view list sorted in order of estimated size.

(Note: an "iteration" in this context is a computation round limited by the availability of main memory). Experimental results showed some performance improvement on one to five processors, but no advantage beyond this point. Failure to exploit smaller intermediate group-bys, coupled with the overhead of independently hashing each cuboid, clearly limited the potential of the approach.

Another hash-based relational research effort was described by Muto and Kitsuregawa in [84]. In this case, the authors proposed a more efficient parallelization technique that used a minimum cost spanning tree constructed specifically for hash-based cube computation (Note: the spanning tree itself was described by the authors of the PipeHash). Specifically, their approach was to partition individual views on a given dimension — in a manner similar to Goil and Choudhary — and then independently compute child view partitions using hash tables constructed from the smallest available parent cuboid. Workload imbalances — caused by skew and data clustering — would be *dynamically* resolved by migrating partitions from busy processors to idle processors. However, the authors provide no physical implementation of their design and instead described *simulated* results. Given the complexity of parallel data cube implementations — particularly ones that do dynamic load balancing — it is hard to evaluate this method without true implementation results. Specifically, their suggestion that all communication would be "free" since it could be completely overlapped with computation is unlikely to be borne out in practice due to the interdependencies between cuboids. As well, they develop an oversimplified data model in which (a) the cardinalities of all dimensions are large (making partitioning much easier) and (b) partitions never have to be split (a likely occurrence in practice that would necessitate the eventual merger of partition segments). Further, we note that they make no mention of the costs associated with the required $O(d)$ re-partitioning rounds. Finally, they appear to have underestimated the effect of data sparsity, the result being

a significant understatement of the hashing costs associated with views in higher dimensions. The near optimal experimental results are consequently of little practical value. And though they did suggest that a true implementation of the algorithm was forthcoming, none appears to have been presented in the literature. It is likely that a working design of this algorithm would require significant augmentation.

In [86], Ng, Wagner and Yin describe four separate algorithms designed specifically for fully distributed PC-based clusters and large, sparse data cubes. The first two techniques are based upon the BUC design of [10], and as the name would suggest, both construct cubes in the direction of coarse granularity cuboids to fine granularity cuboids. In the first case, algorithm RP (Replicated Parallel BUC) takes the lattice and carves it into d sub-trees where each sub-tree corresponds to a collection of views containing the same attribute. For example, exactly half of the views will contain attribute A, half of the remaining views will contain B, and so on. Parallelization is achieved by simply distributing the unevenly sized sub-trees across the network in a round robin fashion. If more than d processors exist, the extras sit idle during the computation. Not surprisingly, the coarseness of the partitioning produced very poor performance results. The authors try to improve upon this situation with algorithm BPP (Breadth First Writing, Partitioned Parallel Buc) . Here, data is partitioned across all processors. To avoid the sometimes excessive communication costs associated with the $O(d)$ merge phases required by the Goil and Choudhary design, they in fact create d distributed copies of the entire fact table, one for every dimension. However, performance and load balancing results were only marginally better than RP, largely because the costs associated with computing partitions of equivalent size tend to vary widely (due to skew and clustering patterns in the data set). Failure to incorporate this information into the workload scheduler will (and did) result in unpredictable performance.

In an attempt to create a more responsive scheduling mechanism, the authors

designed a pair of algorithms more in keeping with the top down design methodology. Here, a *dynamic* scheduler is employed. Specifically, a master scheduler dictates which tasks — individual views or clusters of views — are assigned to given processors *at runtime*. As with RP, each node contains its own copy of the full fact table. The first of the two algorithms, ASL (Affinity SkipList), decomposes the lattice into its 2^d individual components, and distributes them one by one to the "best" processor, where the best processor is the one associated with an already computed cuboid possessing greatest affinity to the candidate view. Here "affinity" essentially refers to the existence of common attributes. The second algorithm PT, (Partitioned Tree), recursively divides the lattice into subtrees — partitioned on a particular attribute — and assigns each sub-tree cluster to an available node where it will actually be computed with the BUC algorithm. In making its scheduling decisions, the scheduler tries to exploit any affinity that exists between the root of the current sub-tree and the nodes already assigned to the target processor. Experimentally, the finer granularity scheduling of the two algorithms provided better load balancing and, in turn, better overall performance than their bottom up counterparts. Good speedup was obtained up to eight processors. However, it declined very quickly after this point. The reason for the performance degradation is that the type of scheduling used by these algorithms does not capture the "global" cost information of the complete lattice, as did for example the PipeSort spanning tree. As a result, only limited cost reductions can be obtained when the workload is highly distributed since computation of the localized view subsets is poorly coordinated.

3.3 Motivation

Though the research efforts described in the previous section have attempted to exploit the power of multi-CPU systems to more efficiently compute the data cube, they have been only a partial success. Each provided performance results that were either

unimpressive or suggested that scalability would in fact be quite poor. The most sophisticated of these designs (Goil and Choudhary) represents a MOLAP based system that is tied to proprietary data structures and database management systems. For these two reasons — performance and accessibility — there is clearly an opportunity and an impetus for further research in this area.

In approaching the design of new parallel data cube algorithms, we identified the following six primary objectives:

1. Build upon proven, optimized sequential algorithms for local computation.
2. Exploit well studied problems in the parallel computing literature for the purposes of workload distribution.
3. Minimize the communication costs due to view re-distribution.
4. Employ global costing information to ensure that local computation is partitioned/balanced as equitably as possible.
5. Support straightforward integration with standard relational systems.
6. Minimize the complexity — data structures, algorithms, resource management — that would likely lead to an unworkable practical implementation.

In the remainder of this chapter, we present the details of a new parallel algorithm for full data cube computation. Upon the conclusion of the chapter, we will review the extent to which we have accomplished each of the six basic objectives.

3.4 A New Approach to Parallelizing the Data Cube

In designing a new parallel data cube algorithm, a number of preliminary issues had to be addressed. What would be the target architecture? Of the many sequential algorithms available, which would be most amenable to a parallelization effort? And,

finally, how should the workload be partitioned across processors? In this section, we answer these three fundamental questions, and provide the details of the algorithm that grew out of this analysis.

3.4.1 The Target Architecture

In Appendix A we note that contemporary parallel architectures tend to fall into one of two categories: distributed memory and SMP-based. Either could serve as the basis of attempts to design practical parallel data cube algorithms. We have in fact opted for a distributed memory model for the following reason. Specifically, distributed memory designs can generally be mapped to SMP architectures since SMP system builders typically provide efficient mechanisms for running MPI-based code on a shared memory system. Since no such support exists for mapping in the other direction (i.e., from OpenMP-based SMP to distributed memory), then a distributed memory implementation is likely to be portable to a much wider range of parallel machines.

While the processor/memory architecture is the most fundamental means by which to classify parallel machines, it is not the only one. In particular, we may also analyze such architectures in terms of their I/O models. Modern parallel machines may have either of two basic designs: (i) distributed disk or (ii) shared disk. Distributed disk is the most intuitive model. Here each processor is associated with one or more local disk units which can not be directly accessed by other processors. Any disk-to-disk transfers must be accomplished by way of the shared communication fabric. Shared disk, on the other hand, stores data on one or more *disk arrays*. A disk array is a single *logical* storage device that is able to *stripe* files across multiple storage platters in order to improve read/write times. Moreover, when employed as a storage component for parallel computers, the array supports I/O operations from any of the processors in the network. While the variety of storage and striping technologies is

significant, in the current context it is necessary to understand only that independent processors/processes may treat the array as if it were a local storage medium.

We have chosen to target both distributed disk and shared disk with our new algorithm. The distributed disk environment is an obvious choice in that the exploitation of a collection of simple, commodity-type disks makes this platform quite cost effective. In turn, the affordability makes it extremely common. We note, however, that there may be times when creating multiple copies of the fact table may not be feasible. For the full data cube problem, this may not be an issue. Specifically, because the data cube output is much larger in size than the original fact table (often hundreds of times or more), then the additional storage required for a local copy of the fact table is not likely to be a serious concern. For the partial cube problem that we will study in the next chapter, however, it is entirely possible that the subset of views on each local node would be smaller in size that the fact table. Moreover, if the fact table is quite large and resources are at all limited, than a single copy of the fact table may be the only option for these partial cube implementations. For this reason, we felt it was important to design an algorithm that would be equally at home in either environment. In doing so, we guarantee that the parallel data cube algorithm described in the remainder of this chapter — and the partial cube extensions presented in Chapter 4 — are well-suited to today's practical data warehousing problems.

3.4.2 A Sequential Base

While the literature describes a significant number of sequential data cube generation algorithms, we have chosen to build upon the sequential PipeSort as the basis of our parallel implementation. In fact, there are a number of reasons for this selection.

- Sort-based data cube algorithms have been shown to outperform the hash-based alternatives [105, 86].

- Sort-based models can be extended gracefully to handle very large data sets. In other words, the costing metrics need only be adjusted to reflect the new costs associated with external memory sorting.

- While other algorithms may perform better on specific data sets (e.g., BUC in very sparse spaces), the PipeSort provides acceptable performance on a broad range of problems.

- Because the PipeSort is based directly upon a graph representation of the lattice, it is possible to explore graph partitioning algorithms for workload distribution. In fact, this type of distribution mechanism is relatively common in the parallel computing literature.

- The PipeSort manipulates relational tables rather than proprietary database structures.

For these reasons, the PipeSort was an appropriate choice for an initial implementation. We note, however, that from a design perspective, the workload partitioning model described later in this chapter is adaptable to other sequential algorithms. While we have not pursued such alternatives, it should be understood that our algorithmic framework may be viewed as both a physical design and a conceptual model.

Finally, we note that in our current research we restrict our attention to internal memory implementations. This is an unavoidable compromise given the prohibitive time requirements for the development of optimized parallel external memory sorting sub-systems. However, as described in Item 2 above, algorithmic extensions to external memory are quite straightforward. All that is required — besides the external memory code itself — is an augmentation of the costing model for the larger external memory sorts. As such, the reader should understand that any restrictions on input size shall be seen as purely the result of practical limitations, *not* algorithmic limitations.

3.4.3 Partitioning for Parallel Computation

We begin then with a sequential PipeSort algorithm that generates a minimum cost spanning tree MCST from the cuboid lattice. Since the PipeSort manipulates a weighted tree, our primary objective when considering a parallel version of the original algorithm is to devise a partitioning approach that supports an equitable distribution of work, given the costing complexity associated with cuboids of radically different sizes. One approach, taken in [48, 47, 49, 86], is to partition individual views such that each is partially computed by each one of the p processors in the system. However, as we saw in Section 3.2, this form of view partitioning can introduce large communication costs. Moreover, this option is exceedingly complex in practice due to the fact that for $1 \leq i < j \leq p$, and a data set distributed as A_1, A_2, $\ldots A_p$, the size of partition A_i may be significantly different than that of A_j due to the characteristics of the data set. Instead, we opt for a partitioning strategy in which a *master* processor generates p independent subproblems from *sub-trees* of the MCST, each of which can be solved by an independent *compute* processor using a sequential pipeline algorithm. There are two chief advantages. First, by partitioning the lattice in advance, we are able to exploit all available costing information to make *global* costing decisions. Second, because individual cuboids will be constructed in their entirety on a particular node, no view re-distribution or merge costs are required. In fact, the only node-to-node communication is the single *task list* that the master node sends to each compute node.

Since determining an optimal partitioning of a weighted graph is NP-complete (see Section 3.8), a heuristic approach which generates p subproblems with "some control" over the sizes of those subproblems holds the most promise. With respect to partitioning logic, we clearly want the sizes of the p subproblems balanced. However, we also want to minimize the number of subtrees assigned to a single processor. To understand why this would be the case, note that when a sub-tree S is created, the

root of the new subtree S_{root} cannot be created from its parent P_{org} in the original spanning tree since P_{org} may have been distributed to another processor. Even though the use of a shared disk implies that P_{org} would be accessible to S_{root}, the fact that P_{org} is one of the last cuboids computed in its sub-tree, while S_{root} is the first in its tree, prevents us from exploiting the obvious relationship between the two. As such, S_{root} must be computed from the larger "raw" data set R. We note at this juncture that, in practice, the use of R as a parent for S_{root} is not as large a penalty as one might expect due to the nature of sub-tree generation. Specifically, in almost all cases the sub-trees are generated such that their root node is in the upper levels of the lattice and is consequently not significantly smaller than R. Nevertheless, too many subtrees can result in sub-optimal sorting and I/O performance. The solution we develop balances the impact of partition size and sub-tree count.

Our heuristic makes use of a related partitioning problem on trees for which efficient algorithms exist, namely the *k-min-max partitioning problem.* Definition 1 provides a formal description of the fundamental problem.

Definition 1. *The min-max problem can be defined as follows: Given a tree T with n vertices and a positive weight w assigned to each vertex, delete k edges in the tree such that the total weight $\Sigma_{i=0}^{l} w_i$ of the l nodes in the largest resulting subtree is minimized.*

The k-min-max partitioning problem has been studied in [6, 44, 94]. In our case, we build upon the *Min-Max Shifting Algorithm* described in [6] and presented here as Algorithm 5. This algorithm is based on a *pebble shifting* scheme where k pebbles or *cuts* are shifted down the tree, from the root towards the leaves. The objective is to reduce the size of the largest current partition during each pass. Whenever possible, this partition is minimized by *down-shifting* the cut above its root node to the child edge with the heaviest *down-component* — defined as the weight of a partition beneath a given node or edge. Once done, we walk back up the tree, checking to see if the previous downshift has created an unnecessarily large partition above the original

root. If so, we reduce its size by *side-shifting* cuts. The algorithm continues until the largest partition can no longer be reduced in size.

Algorithm 5 k-min-max Algorithm: Becker, Schach and Perl

Input: A weighted tree T with positive weights assigned to its n vertices.
Output: A set of k sub-trees in which the weight of the largest sub-tree is as small as possible.

```
1: Place k − 1 cuts on the edge incident with the root vertex.
2: while rootPartition is lightest partition do
3:   if largest partition has one or more vacant edges beneath its root node q then
4:     Downshift cut at q to child edge with largest down-component
5:   else
6:     Terminate
7:   end if
8:   repeat
9:     Get the next node w along the path from q towards the root of T
10:    if w has a cut on an incident edge e_i whose down-component < down-
       component of the current vacant edge e_v then
11:      side-shift the cut on e_i to e_v
12:    end if
13:  until a cut is encountered on the path towards the root of T
14: end while
```

We note, however, that min-max algorithms are designed to work on static graphs. In other words, it is expected that the node weights remain fixed during the execution of the algorithm. In our case, that is not true. Recall that when a sub-tree S is extracted from the MCST for the purpose of network distribution, the root node of S cannot be computed from its designated parent but must be computed instead from the "raw" data set. As such, min-max cutting dynamically alters the cost structure of the MCST. Moreover, because any given cut may be shifted many times during the life time of the algorithm, the estimated cost to compute a particular view may oscillate frequently before a "stable" configuration is reached. We deal with this problem by maintaining a *cut hash table*. When a cut is moved to a new edge — whether by a down-shift or side-shift — the reference (i.e., pointer) to the target edge is used as

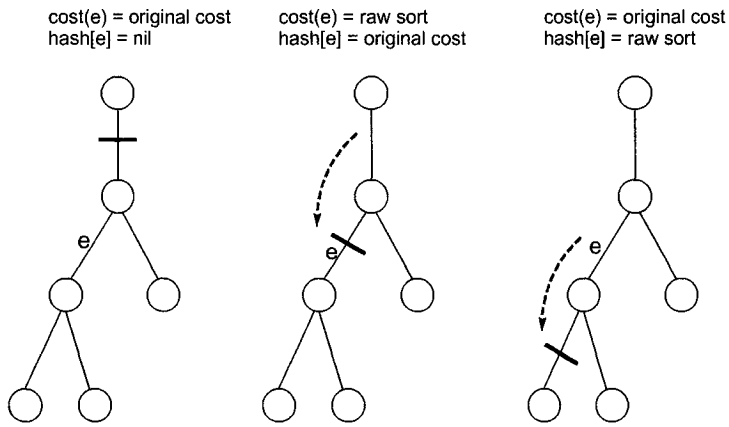


Figure 3.1: The use of a "cut" hash table to support dynamic min-max.

a hash key to determine whether that edge has previously "hosted" a cut. If so, the value from the hash table and the current edge value are swapped. If not, the cost of sorting the "raw" data set replaces the current cost which is inserted into the hash table as the value associated with the current edge. We refer to this extended form of the shifting algorithm as *dynamic k-min-max*. An illustration of the algorithm's adaptive costing can be seen in Figure 3.1.

We should point out that the inclusion of the cut hash table has no effect upon the algorithm's ability to terminate in a stable state. During each iteration, a down-shift is always performed. Since these shifts are in one direction only and, moreover, are finite in number given the fixed radius of the tree, the use of dynamic costing cannot introduce *looping* behavior. The algorithm *must* terminate, though of course the partitioning decisions may be quite different.

3.4.4 The Parallel PipeSort Algorithm

Returning to the partitioning problem itself, we note that because the goal of dynamic k-min-max is to minimize the weight of the largest sub-tree, the algorithm does not necessarily result in a partitioning of T into subtrees of *equal* size, nor does it address tradeoffs arising from the number of subtrees assigned to a processor. Consequently,

we use tree-partitioning as the initial phase of a two-part strategy. To achieve a better distribution of the load we apply an *over sampling* mechanism: instead of partitioning the tree T into p subtrees, we partition it into $s \times p$ subtrees, where s is an integer, $s \geq 1$. Then, we use a *packing heuristic* to determine which subtrees belong to a given processor. Essentially, our packing heuristic considers the weights of the subtrees and uses those weights to combine distinct trees so as to balance the cost of computation across nodes. It consists of $s - 1$ matching phases in which the p largest subtrees (or groups of subtrees) and the p smallest subtrees (or groups of subtrees) are paired up. In the end, s subtrees are assigned to every processor. Details are described in Algorithm 6.

Algorithm 6 Tree Partitioning

Input: A spanning tree T of the lattice with positive weights assigned to the nodes (representing the cost to build each node from it's ancestor in T). Integer parameters s (over-sampling ratio) and p (number of processors).

Output: A partitioning of T into p subsets $\Sigma_1, \ldots, \Sigma_p$ of s subtrees each.

1: Use dynamic min-max to compute an $s \times p$-partitioning of T into $s \times p$ subtrees $T_1, \ldots, T_{s \times p}$.
2: Distribute subtrees $T_1, \ldots, T_{s \times p}$ among the p subsets $\Sigma_1, \ldots, \Sigma_p$, s subtrees per subset, as follows:
3: Create $s \times p$ sets of trees named Υ_i, $1 \leq i \leq sp$, where initially $\Upsilon_i = \{T_i\}$. The weight of Υ_i is defined as the total weight of the trees in Υ_i.
4: **for** $j \leftarrow 1$ *to* $s - 1$ **do**
5: Sort the Υ-sets by weight, in increasing order. W.l.o.g., let $\Upsilon_1, \ldots, \Upsilon_{sp-(j-1)p}$ be the resulting sequence.
6: **for** $i \leftarrow 1$ *to* p **do**
7: Set $\Upsilon_i = \Upsilon_i \cup \Upsilon_{sp-(j-1)p-i+1}$
8: Remove $\Upsilon_{sp-(j-1)p-i+1}$
9: **end for**
10: **end for**
11: **for** $i \leftarrow 1$ *to* p **do**
12: Set $\Sigma_i = \Upsilon_i$
13: **end for**

The tree partitioning algorithm is embedded into our parallel data cube construction algorithm. Algorithm 7 presents the basic model. Like the sequential algorithm, the lattice must be augmented with appropriate costing values. First, the final size of the 2^d proposed cuboids must be estimated. Then, using these figures, we develop a complete pipeline costing framework — including sorting, scanning and input/output estimates — that will eventually allow the bipartite matching mechanism of the PipeSort to determine the most cost effective means by which to move from $level_i$ to $level_{i-1}$ in the lattice. Once the minimum cost spanning tree has been extracted in Step 3, we use dynamic k-min-max with over-sampling to determine the appropriate sub-tree distribution across the p processors. Finally, when the local processors have received their task lists, a local pipeline computation algorithm generates the assigned output views to complete the distributed data cube computation.

Algorithm 7 Parallel PipeSort

Input: A data set R, with each of its n records composed of d feature attributes and one measure attribute.

Output: A collection of 2^d cuboids, each presenting a summarized view of a unique subset of the d feature attributes.

1: On the master node M, apply a storage estimation method in to determine the approximate sizes of all 2^d cuboids in the data cube lattice L.
2: Augment L with the estimated costs of sorting and scanning each of its 2^d nodes.
3: Using the sequential PipeSort algorithm, extract a minimum cost spanning tree T from L.
4: Execute Algorithm *Tree-partition*, creating p sets $\Sigma_1, \ldots, \Sigma_p$. Each set Σ_i contains s subtrees of T.
5: **for** $i \leftarrow 1$ *to* p **do**
6: Distribute Σ_i to p_i
7: **end for**
8: Each processor p_i, $1 \leq i \leq p$, performs the following step independently and in parallel:
9: Compute all group-bys in subset Σ_i using the sequential pipeline computation algorithm.

3.5 Optimizing Performance

Though the primary focus of our parallel data cube research has been upon optimizing the load balancing characteristics of the parallel computation, it should be noted that final performance is ultimately limited by the efficiency of the underlying sequential algorithms. Within the PipeSort context, this would suggest that the algorithm for constructing each of the 2^d cuboids deserves special consideration. We note, however, that while the PipeSort authors described the algorithm for spanning tree extraction in some detail, they provided very little information on the computationally expensive *pipeline* phase. So though it is true that the views of a given pipeline can be constructed in a single pass of some sorted input set, it not at all obvious that this can be done as efficiently as one might expect.

In this section we discuss in detail the fundamental issues that need to be addressed in constructing an efficient PipeSort-based system. Specifically, we will examine the parameters, goals, and logic of the pipeline algorithm to determine whether variations in algorithm design or implementation could tangibly affect runtime performance. Our analysis lead to the identification of four *performance limiters*, components of the pipeline algorithm whose individual design *could* significantly impact sequential runtime. The limiters are listed below:

1. **Sort Quality**. The first step in pipeline processing is a sort of the input view. Since $\begin{pmatrix} d \\ \lceil d/2 \rceil \end{pmatrix}$ is a lower bound on the number of sorts required to construct the full cube, a reduction in sorting costs will be central to improving overall performance.

2. **Data Movement**. The PipeSort algorithm is data intensive. Because $O(2^d)$ views may be created, many of which will have entirely different attribute orderings (necessitating record permutations), the physical movement/copying of records and fields can become extremely expensive on large, high dimension

problems.

3. **Aggregation Operations**. At the heart of the pipeline process is the aggregation of partial totals into records of varying granularity. The existence of unnecessary or redundant aggregation operations results in a needless slowdown in pipeline execution.

4. **Input/Output Patterns**. The PipeSort algorithm moves enormous amounts of data to/from disk. In such environments, disk head movement and buffer flushing are a serious performance issue.

In the remainder of this section, we discuss our approach to the optimization of each of the performance limiter categories. In Section 3.9, we demonstrate how these optimizations result in an order of magnitude improvement in performance.

3.5.1 Optimizing Sorting Operations

Sorting can perhaps be viewed as the single most important algorithm class in the collective fields of computer science. Not surprisingly, the literature identifies a broad collection of algorithms for placing objects into ordered sequences. That being said, practical sorting implementations are typically associated with a small number of algorithms such as merge sort and Quicksort [22]. The latter, in fact, is almost always a standard component of language and operating system libraries.

Most generic sorting algorithms have one major feature in common — they are *comparison based*. In other words, records are arranged by comparing their values to other records in the input set. In terms of cost complexity, such algorithms have a lower bound of $\Omega(n \log n)$ [22]. In contrast, non-comparison sorts have also been developed. With an $\Omega(n)$ lower bound (we must at least look at every record), they avoid record-to-record comparisons by exploiting some type of *a priori* knowledge with respect to the nature of the data set. Such information, for example, might be

associated with the domain of the key to be sorted or the uniformity of its distribution.

In [10], the authors of the BUC method use an algorithm called *Counting Sort* [22] to reduce the cost of partitioning. Counting sort is an $O(n + k)$ algorithm, with k equivalent to the range of the key to be sorted, that determines the exact position of a key in the output array without relying on key comparisons. In particular, it maintains a table of the number of occurrences of each distinct value within the domain of a sort key, then uses this information to determine exactly how many records should be found *before* this value in the sorted output array. We note that when k is $O(n)$, the upper bound for Counting Sort reduces to $O(n)$. In the OLAP context, k is equivalent to the cardinality of the dimension to be partitioned. Because the number of *distinct* values within a dimension A is typically much smaller than the total number of records n in the fact table, the use of Counting Sort can provide linear time sorting functionality for the BUC method.

Counting Sort was applicable to the BUC algorithm because BUC's bottom-up model relies upon single attribute sorts. Our top-down PipeSort, however, requires multi-dimensional sorting. While the Counting Sort cannot be utilized directly in this context, we can extend its sorting model to suit such an environment. Specifically, we can utilize a *Radix Sort*, a multi-pass extension of the Counting Sort. Algorithm 8 describes the technique. Working in *reverse* order — from least significant digit to most significant — we perform a series of k counting sorts on each of the k relevant (possibly non-contiguous) attributes of the input set. We note that this technique works only because the underlying Counting sort is *stable*. Stability refers to the fact that elements with the same value appear in the same relative position in the output list as in the input list. With respect to running time, the k-iteration Radix Sort may be bounded as $O(kn)$.

Nevertheless, because of the k passes, it is not always the case that a Radix Sort will be preferred over a well-optimized Quicksort. Proposition 1 suggests how

Algorithm 8 Data cube Radix Sort

Input: A d-dimensional input set, and a set of k attributes to sort.
Output: An output set, sorted on the k dimensions.
1: **for** $i \leftarrow k$ to 1 **do**
2: Perform a Counting Sort on attribute i
3: **end for**

the performance comparison should be made. Experimental evaluation has in fact confirmed the accuracy of this observation.

Proposition 1. *For a data set of d dimensions and n records, a k-attribute sorting should be performed with a Radix Sort rather than a QuickSort if $3k < \log n$.*

Proof. Given the $O(kn)$ runtime of the Radix Sort and the $\Theta(n \log n)$ bound on the Quicksort, an initial analysis would suggest that for $k < \log n$, the Radix Sort would be more cost effective. For practical implementations, however, we must consider the constants associated with each sort. While overhead is very low with Quicksort, we note that each iteration of Counting Sort makes two passes of the input set and a single pass of the intermediate k-record table. With k bounded as $O(n)$, we therefore see that a Radix Sort is likely to be faster only if $3k < \log n$.

□

Our pipeline sorting mechanism therefore dynamically determines the appropriate sort for a view of a given size and dimension count by evaluating this expression for each input set. We note that while Radix Sort offers relatively little benefit for small problems, it becomes an increasingly valuable algorithmic component as the data sets become very large. Furthermore, for the partial cube problems that will be discussed in the next chapter, the Radix Sort is especially appealing in that we often see large, high dimension input sets being sorted into pipeline views with significantly lower dimension counts — a situation ideally suited to Radix Sort.

3.5.2 Data Movement

Given the 2^d views generated by a full cube computation, and the varied orderings of attribute values for each pipeline, the movement of data is a significant concern for an efficient PipeSort implementation. In fact, we can identify two distinct algorithm components that require the algorithm designer to address this concern. In the first

case, we note that both Quicksort and Radix Sort (as well as most other sorting algorithms) perform a large number of key movements during the course of execution. Quicksort moves keys after each of its $\Omega(n \log n)$ comparisons, while Radix Sort must write the input list to a sorted output list following each of its k iterations. More importantly, however, data cube *keys* are actually multi-dimensional records. As such, a massive amount of data copying must be performed since $d + 1$ fields (the complete input record plus its measure attribute) must be moved every time.

In the second case, data movement is associated with pipeline processing. Because the pipeline aggregation process relies upon a prefix ordering that is by definition different from that of the input set (this is why a re-sorting is required), then the order of the attributes in the records of the input set must be permuted to that of the parent view in the pipeline before aggregation can be performed. We note that while we could permute the input set by copying it to a fresh buffer, this would effectively double memory requirements for the application. A more practical "in place" copy would require two complete data movements — one to permute each record into a clean buffer and one to copy it back to the original location. Total transfer cost would therefore be $2n(d + 1)$.

Our goal in minimizing data movement is to eliminate record copying once the input set has been read into memory. To accomplish this objective, we have designed the algorithm to allow it to exploit *data indirection.* By indirection, we mean that instead of manipulating data directly, we instead manipulate *references* to that data. Within the pipeline algorithm, there are actually two distinct forms of indirection — and consequently, two types of references. In the first case, we utilize *vertical* or record-based indirection. Here, we maintain a table of pointers (memory addresses) to the records in the sorted input set. For any operation that would require a record as input, that record is obtained by following a pointer to the record's location in the original input buffer. We note that because standard array offset calculation —

for example, $myArray[i]$ — actually involves a conversion from a subscripted array expression to a physical memory address, access by vertical indirection is actually slightly faster than conventional access.

In the second case, we employ *horizontal* or attribute-based indirection. Here, we attempt to largely eliminate the costs associated with permuting attribute values within individual records. To do so, we maintain an *attribute indirection table* of size k that maps the *intended* position of each of the k attributes in the pipeline parent view to its *actual* position in the original record. Each pipeline has its own unique map. The cost of horizontal indirection is an additional offset calculation for each attribute access. We note, however, that during pipeline computation, the heavily accessed mapping table is likely to be held in the high speed L1/L2 processor caches, dramatically reducing offset calculation costs. Data permutation, on the other hand, generates true core-memory to core-memory swapping. Experimental evaluation has shown that the cost of a full permutation of the input set is at least an order of magnitude more expensive than that of horizontal indirection.

Figure 3.2 graphically illustrates the use of both forms of indirection. A two dimensional attribute reference is identified by first following the appropriate pointer into the input buffer and then mapping the dimension reference to the proper position in the record array. We note that although the input set does not *appear* to be sorted — in fact, the input buffer has been completely untouched since it was first loaded into memory — it is nonetheless ordered as per the requirements of the application (in this case by the attribute order defined in the attribute indirection table). The reader may convince themselves of this by following the pointers and comparing dimensions.

Note that both the sort routines and the pipeline methods utilize both vertical and horizontal indirection. Radix Sort and Quicksort are both implemented so as to work with record references. Changes to record positions — whether on the basis of record comparisons (Quicksort) or counting-oriented re-orderings (Radix Sort) are

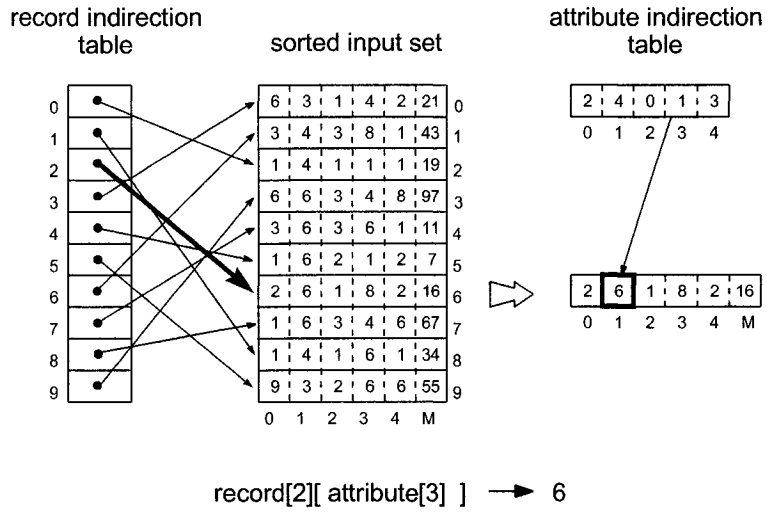


Figure 3.2: Resolving an attribute reference by way of vertical and horizontal indirection.

reflected by changes in reference positions. Once the input set has been (indirectly) sorted, the record indirection array is passed to the pipeline processing module as input. The pipeline code makes a linear pass through the reference table, using the attribute mapping table to identify values relevant to each granularity level in the pipe.

Because the costs of indirection are negligible in practice, significant runtime reductions are achieved. Theorem 1 formalizes this notion for the case of the Radix Sort. An analysis for QuickSort could be performed in a similar fashion.

Theorem 1. *For an input set of n records with k attributes to be sorted with a Radix Sort, vertical and horizontal indirection eliminates up to* $n(k^2+2k+2)$ *attribute copies from* each *pipeline computation.*

Proof. Data movement for Radix Sort occurs only after each of the k Counting Sort iterations and involves a single copy of the n input records to a fresh output buffer. We can therefore bound the movements due to sorting by the k iterations on the $n(k+1)$ attribute values. In terms of data set permutation, recall that $2n(k+1)$ copies are required. Total possible data movement for what we will call the (S)tandard pipeline is therefore the summation of the two expressions:

$$T_S = (k * n(k+1)) + (2 * n(k+1))$$

$$= nk^2 + nk + 2nk + n$$

$$T_S = n(k^2 + 3k + 2)$$

Since the data movement using (I)ndirection is limited to the movement of pointers (single 32-bit words) during each of the k Counting Sort passes, then the movement due to indirection T_I is equivalent to kn. The savings due to indirection is therefore:

$$T_{DIFF} = T_S - T_I$$

$$= n(k^2 + 3k + 2) - kn$$

$$T_{DIFF} = n(k^2 + 2k + 2)$$

□

We note that Theorem 1 describes data movement in terms of an upper bound. A lower bound is not provided due to the possibility that the input set could be pre-sorted, in which case no data movement would be required. However, such situations would rarely be encountered in practice.

Proposition 2. *Pre-sorted input can be associated with at most one pipeline.*

Proof. The construction of a pipeline u produces a set of views whose records are sorted according to a specific prefix order. Any of these views may later be used as the input set for some other pipeline v. By definition, the prefix order of v must be different than the order of u. Otherwise, the views of v would have been part of u since they would have shared a common prefix. As such, it is not possible for any view of u to be pre-sorted in the order required for any view in v. In fact, only the fact table itself can have this property and, even then, can only be pre-sorted in the order required for a single pipeline.

□

We may apply Theorem 1 to a practical case to give some sense of the potential performance impact. Consider, for example, an eight dimensional data set with 10 million records. Using a Radix Sort and indirection, the upper limit for data transfer costs is 8×10^7. In the absence of indirection, the potential cost balloons to 8.2×10^8, a difference of over 70,000,000 operations.

3.5.3 Aggregation Operations

Once the sorting phase has been completed, the input set is passed to the pipeline processing module. As per the description of the original PipeSort paper [105], only a single pass of the input set is required to generate each of the m views in a given pipeline. As previously noted, however, the details of a pipeline implementation were never provided. In fact, even though this component of the algorithm is asymptotically $O(n)$, the constants within each of the n iterations can be very large, with the result that a naively implemented pipeline algorithm may be unacceptably expensive in practice.

To see why this might be the case, we need to examine what happens during each iteration. The algorithm compares two records during each step — the current record R_i and previous record R_{i-1}. It must examine the record R_i to find those changes in attribute values that might affect any of the m cuboids in the current pipe. For a given attribute d, a change in its value might or might not result in the creation of a new output record on a view V_i, where $1 \leq i \leq m$, depending on whether or not d is actually a component of V_i. Checking each of the d attributes to determine if it affects any of the up to d-attribute views in the m-length pipe produces a $O(d^2m)$ run time for the record checking phase. Doing this for each of the n input records quickly begins to erode the perceived benefit of a $O(n)$ algorithm.

The number of aggregation operations actually performed is another concern. Because we are iterating through n records, each of which will contribute its measure value to the subtotals of each of the m views, there is the potential for $n*m$ aggregation operations. For large n and large m — and multiple pipelines — the amount of CPU time dedicated to aggregation can be large.

Algorithm 9 presents a new pipeline algorithm that has been designed to address these concerns. In Line 1, we begin by initializing the buffer for the parent view in the pipeline. We then move on in Line 2 to the main loop that will examine each of

the n records for attribute changes. Lines 4-8 simply look for the first attribute — working from the most significant to the least significant of the attributes relevant to the current pipeline — on which there is a change. If there has been no change, the measure attribute is simply aggregated into the buffer of the parent view and we move on to the next record. However, if there has been a change, we use the *change position* j as an implicit indicator of *all* affected cuboids. For example, given that attribute checking proceeds from coarse to fine granularity in the prefix order, we know that a view with at least l feature attributes *must* be affected if $l >= j$. Conversely, for a "coarse" view with less then j attributes, we can be guaranteed that a change has not occurred on any of its attributes. We may therefore skip any further processing on this view. For each of the n records, then, the total number of conditional checks is simply the number required to find j, plus the number required to identify the affected cuboids. This $O(d + m)$ Pipeline Aggregation method is significantly more efficient than the naive $O(d^2m)$ design described above.

Returning to the issue of the number of aggregation operations, recall that $O(mn)$ summations may be required. In Lines 13-15 of Algorithm 9, note that aggregation only occurs on any of the $m - 1$ views beneath the parent view when a change in attribute value has occurred on the *next coarsest* view. In other words, aggregation on these views is never done from the records of the input set. Instead, we use the knowledge that a change has occurred on view V_{i+1} to *trigger* an update on the running sub-total of V_i. We refer to this approach as *lazy aggregation*, a term indicative of the fact that we hold off on aggregating into V_i until absolutely necessary. In fact, for a given input data set and its associated pipeline, it is possible to show that lazy aggregation results in the optimal/minimal number of aggregation operations. Definition 2 and Theorem 2 formalize this notion.

Definition 2. *We define a* ***touch*** *on an input set as an access for the purpose of aggregation.*

Algorithm 9 Pipeline Aggregation

Input: A sorted input set of n record and d dimensions, and an ordered set of k dimensions that form a pipeline prefix order.

Output: A set of m output views, each aggregated at a different level of granularity.

```
1: Initialize parent view output buffer with contents of first input record
2: for i ← 1 to n do
3:    Get pointers to CurrentRecord and LastRecord
      {Find position of first change in current record}
4:    for j ← 1 to k do
5:       if CurrentRecord[j] != LastRecord[j] then
6:          break
7:       end if
8:    end for
      {Enter aggregation phase}
9:    if j == k then
10:      Aggregate measure value into buffer of parent view
11:   else {perform lazy aggregation}
12:      for all views m in pipeline, running from finest to coarsest do
13:         if j < number of attributes in m then
14:            Subtotal for view[m - 1] += subtotal for view[m]
15:            Create new aggregation record for view[m]
16:         else {no more aggregation required on this input record}
17:            break
18:         end if
19:      end for
20:   end if
21: end for
```

Theorem 2. *For a pipeline P, of length m, lazy aggregation results in the minimum possible number of touches.*

Proof. Each of the m cuboids in the pipeline is computed record-by-record during a linear pass over the input set. For each such view in the pipeline, we aggregate records at increasingly finer levels of granularity. For a view V_i, $1 \leq i < m$, its records can be seen as an aggregated summary of the records from any of the views above it in the pipeline. Note that the most concise summary of the records of V_i is found in V_{i+1}, its immediate parent in the pipeline. Use of this parent view would allow V_i to be computed strictly from the j records of V_{i+1}. In other words, V_i would be computed in just j touches. Now, note that during lazy aggregation, the computation of V_i only produces a touch — on V_{i+1} — when a change in attribute value occurs on the view V_{i+1}. Since each such change is associated with the creation of exactly one new record in V_{i+1}, we therefore conclude that the number of touches generated for view V_i by lazy aggregation is exactly equivalent to j, the number of records found in the most concise aggregation summary available to V_i.

□

Again, a practical example may be used to illustrate the potential performance gain. Consider a data set of one million records, and a collection of k attributes, each with a cardinality of ten. For a view A, the naive form of aggregation would result in $n = 1,000,000$ distinct aggregation operations. With lazy aggregation, however, summing into A would only occur when triggered by record creation on its immediate parent, say AB. Since the size of AB is bounded by its cardinality product $\prod_{i=1}^{d} C_i$, we can see that the total number of aggregation operations required to compute A in this example is just 100.

3.5.4 Input/Output Patterns

The preceding three issues have been associated with computational performance. In this section, the focus shifts to I/O issues relevant to the view construction phase. We note that PipeSort is a very I/O intensive application. Specifically, in high dimensions the output size grows to hundreds of times larger than that of the input. In this context, careful treatment of I/O issues can lead to significant overall performance gains. Conversely, without adequate attention to the execution and ordering of disk

accesses, particularly write operations, the benefit of algorithmic optimizations may be seriously undermined.

We have identified two areas in particular that have the potential to affect PipeSort performance. The first is the problem of *disk thrashing*, a phenomenon that occurs when the operating system tries to do a series of small writes to a set of distinct files stored at different locations on the hard disk unit. Because the time taken to move the disk head to a non-contiguous block is at least an order of magnitude larger than the time taken to simply read the next contiguous block, a series of small write requests can result in significant disk head movement but little actual writing. The pipeline component of the PipeSort algorithm exhibits exactly this type of structure as each of the n iterations in the pipeline loop has the potential to generate m output records, each bound for a different file.

Modern operating systems attempt to limit the effect of this problem by *buffering* write operations. In effect, they delay the writing of small output streams in the expectation that more data may be forthcoming. At some later point, the collective element set may be written with a single request. Given that the operating system knows nothing about the characteristics of the PipeSort application, however, it will likely still thrash to some degree since it doesn't know how long to delay the writes and will often flush buffers to disk before it needs to. If it delays the writes too long, on the other hand, we are presented with a second major problem. The operating system's caches eventually become so saturated that a massive amount of "house cleaning" is required to return the OS to a productive state. This wholesale flushing is extremely resource intensive and is a serious drain on performance. Worse yet, because the state of the OS caches depends upon the existence and requirements of other non-PipeSort applications that may be running on the system, the effect is largely non-deterministic and may impact the application to varying degrees at different times.

We address these issues with the introduction of an I/O manager specifically designed for PipeSort. Embedded directly within the PipeSort framework, the job of the I/O manager is to take over control of input/output operations from the operating system. In so doing, the application controls the size and rate of disk transfers and can tune the flow to largely prevent disk thrashing and cache overload. Figure 3.3 illustrates the I/O subsystem. For a pipeline of length m, the I/O manager controls m "view blocks", each associated with the resources required to manipulate one cuboid. One of these resources is an output buffer that houses contiguous chunks of records destined for disk. When a *logical* write is issued, the output record actually goes to the I/O manager which then places it into the next available slot in the appropriate view buffer. If the *current record* pointer is equivalent to the *max record* pointer, then the I/O manager knows that the buffer cannot accept any more records and it will tell the operating system to write the entire buffer to disk. We note that the write request is exactly that — a "request". The operating system may choose to buffer the current block of data, along with zero or more previous buffer arrays, if it has adequate resources. This is perfectly acceptable.

For an output view of size j and an application buffer size of l, the use of an explicit I/O manager reduces the number of write calls — and potential thrashing instances — from j to j/l. However, because the operating system is free to buffer the PipeSort's write calls, it is still entirely possible that it will continue to do so until it is overwhelmed with back-logged I/O requests — our second major concern. To deal with this problem, we introduce a *throttling* mechanism that prevents the operating system from ever reaching this point. Specifically, when all processing for a given pipeline is completed, we issue a system *sync()* call. The sync (or synchronization) call flushes all application I/O streams so that the contents of any operating system buffers are physically written to disk. Because it is a *blocking* call (it does not return until the disk operations are complete), we guarantee that the operating system will

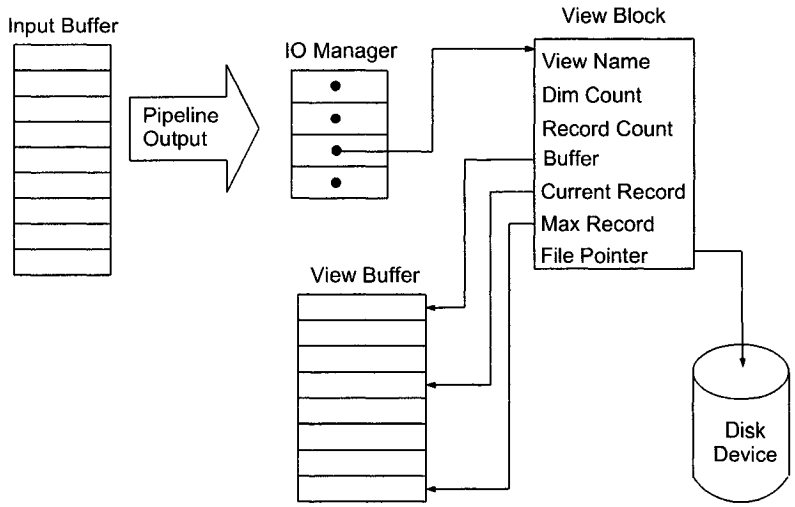


Figure 3.3: An illustration of the data cube I/O manager, showing the resources managed by one of its view "blocks."

never be overloaded with outstanding I/O. In effect, the application uses its knowledge of its own internal structure to appropriately control system resources.

We note, however, that the use of the sync-based throttle mechanism introduces an additional problem. Modern disk controllers employ a feature known as Direct Memory Access (DMA). Instead of requiring the CPU to manage all of the relatively expensive I/O operations, DMA controllers can assume responsibility for moving segments of data directly from memory to disk, thereby allowing the CPU to return to other, computationally demanding tasks. It should be clear that the use of a blocking sync call largely eliminates the benefit of DMA execution since the CPU stands idle as the I/O call executes. More importantly, its use likely erodes the performance gains produced by the algorithmic optimizations already discussed.

We address this issue by recasting the PipeSort implementation as a multi-threaded application. In operating system terminology, a thread is a *lightweight process* which shares the resources of a given application with one or more cooperating threads but

is given its own control registers so that it may execute program instructions independently of the other threads. In the PipeSort, the use of separate I/O and computation threads allows us to enjoy the benefits of DMA without the penalties of blocking sync calls. Specifically, we dynamically generate a new I/O thread at the completion of each pipeline. The sole purpose of this I/O thread is to invoke the blocking sync call. While the new thread waits for the call to terminate, control returns to the original thread which then initiates execution of the *next* pipeline. In this way, we can concurrently process portions of two pipelines — one doing I/O and one doing computation — on a single CPU/DMA architecture. The end result is an I/O system that is well tailored to the application and the resources is runs upon.

3.6 The Costing Model

In previous sections, we have assumed the existence of a costing model that populates the lattice with estimates of cuboid construction costs. Developing an accurate predictive costing model is an important and challenging task. Moreover, it is the accuracy of the cost metrics, as much as any other component of the system, that dictates the quality/balance of the workload distribution. Specifically, if the algorithm does not accurately represent the true cost of constructing pipelines, then there is no guarantee that the decomposed spanning tree will distribute equivalent workloads to each of the processors.

We note that while pipeline costing is a component of the sequential algorithm, its true significance was likely under-appreciated. In particular, costing inaccuracies are somewhat difficult to quantify in the sequential environment unless, for every problem instance, one has a description of the optimal spanning tree for comparative purposes. Since no such optimal result is typically available, it is difficult to judge the accuracy of costing metrics. In parallel, we have the same challenge with respect to the spanning tree but, in addition, poor costing metrics lead to poor load balancing

which, in turn, leads to poor parallel performance. Therefore, even in the absence of an optimal spanning tree, the penalty associated with an inaccurate costing model is both significant *and* obvious.

More importantly, however, we note that the parallel costing model is in fact substantially different than the sequential model. In particular, the sequential framework only concerned itself with the relative costs of sorting and scanning. By incorporating these costs into a bipartite matching algorithm, it was possible to construct efficient pipelines. Note that the designers of the PipeSort did not concern themselves with the I/O costs of the data cube, even though the enormous size of the output makes such costs substantial. They did not do so because in a sequential environment, the I/O costs are fixed. In other words, no matter how a view is created (e.g., ABCD versus DBCA), it always contains the same number of records and, consequently, will always have the same I/O cost. While this is also true in parallel, note that computation is no longer associated with a single processor. Instead, just as each processor will do its own pipeline computation (sorts and scans), it will do its own I/O as well. Moreover, it must be understood that the cost of I/O is not directly proportional to pipeline construction. For example, an expensive sort may be used to build a number of relatively small views, while a cheaper sort may be associated with one or more large views. For this reason, good load balancing can only be achieved if we can incorporate I/O costs into the costing model. In practice, the resulting parallel cost model is significantly more complex than its sequential counterpart.

For illustrative purposes, we break the costing infrastructure into two basic phases. The first deals with the estimation of cuboid sizes while the second takes these size estimates and determines pipeline construction costs. In this section we take a detailed look at both components of the framework.

3.6.1 Cuboid Size Estimation

Since the cuboids that populate the lattice have yet to be created, we must estimate their final sizes. This is in fact a very difficult thing to do accurately in a data cube context since we know nothing about the distribution of data in the fact table. Nevertheless, a number of techniques for size estimation have been described in the literature [58, 110] and will be reviewed in this section. A number of these methods build upon the notion of a *potential space.*

Definition 3. *For a given view V, we refer to the set of all possible values in that view as its* ***potential space****, denoted S_v. If V has k attributes with cardinalities C_1, $C_2 \ldots C_k$, then the size of the potential space is simply the cardinality product $\prod_{i=1}^{k} C_i$.*

3.6.1.1 Cardinality-based Estimation

The simplest cuboid size estimator is the cardinality product itself. Since any point in a k-dimensional view V must be drawn from the potential space S_v, then this product may be used as a crude approximation of the size of the input set. In practice, such a technique tends to grossly overestimate the size of high dimension cuboids. For example, in a ten dimension space with a cardinality of ten on each dimension, and a fact table of one million records, the size estimate for cuboid AB would be 100, likely a reasonably good guess. For the eight dimensional view $ABCDEFGH$, however, our estimate would be 100,000,000, a useless estimate given the size of the original fact table. We can, of course, limit the scale of the inaccuracy by bounding the size of a cuboid as $min(\prod_{i=1}^{d} C_i, size(parent))$, but the basic mechanism is still far too inaccurate for meaningful pipeline computation.

3.6.1.2 Sample Scaling

Another approach is to extract a random sample from the data set, compute a "mini" data cube on this sample, and then scale up the results by the ratio of the size of the fact table to the size of the sample set. The logic here is that the sample provides

a more realistic view of the actual fact table than that provided by the cardinality product. In this case, however, the technique is at least partially defeated by the impact of record sparsity within the sample set. For example, suppose we have a data set of one million records, with each of its ten dimensions having a cardinality of ten. If we extract a sample of 100 records for the purpose of estimating the size of the three dimension cuboid ABC, it is unlikely than many duplicates would be found, given that the potential space has size = 10^3. Let's assume that the size of ABC_{sample} is 90. We would therefore create the scaled estimate as $ABC = ABC_{sample} * (10^7/10^2) = 9 \times 10^6$. This is clearly inaccurate since the potential space is only 10^3. In other words, sample scaling tends to underestimate the number of duplicates that will be found in large data sets.

3.6.1.3 A Probabilistic Method

In [110], the authors propose a view estimation method that builds upon the Probabilistic Counting algorithm first presented in [42]. The Counting-based method works as follows. During a single linear pass over the data set, it concatenates the d dimension fields into bit-vectors of length L and then hashes the vectors into the range $0 \ldots 2^L - 1$. The algorithm then uses a probabilistic technique to count the number of distinct records (or hash values) that are likely to exist in each of the 2^d cuboids. To improve estimation accuracy, a universal hashing function [22] is used to compute k hash functions that, in turn, allow the algorithm to average the estimates across k counting vectors. The end result is a method that can produce very accurate cuboid size estimates (with a bounded error), even for data sets with significant skew.

We have implemented the algorithm in [110] and verified its accuracy. For example, the algorithm produces estimation error in the range of 5–6 % with 256 hash functions. However, its running time on large problems is disappointing. Specifically,

despite an asymptotic bound of $O(n * 2^d)$, the constants hidden inside the inner computing loops are quite large. For the small problems described in previous papers, this is not an issue. In high dimension space, the running time of the estimator extends into weeks or even months.

Considerable effort was expended in trying to optimize the implementation of the algorithm. Despite a factor of 30 improvement in running time, the algorithm remained far too slow. We also experimented with the GNU-MP (multi-precision) libraries in an attempt to capitalize on more efficient operations for arbitrary length bit strings. Unfortunately, the resulting estimation phase was still many times slower than the construction of the views themselves. At this point, it seems unlikely that the Counting-based estimator is viable in high dimension space.

3.6.1.4 Our Own Probabilistic Approach

We have developed our own probabilistic cost estimator for data sets whose point distribution is approximately uniform. We note that, in practice, this assumption will not always be valid. However, for the purposes of algorithm design and evaluation, it serves as a very useful starting point. Moreover, our estimator is quite efficient and, unlike the Counting-based algorithm, can produce estimates in a fraction of the time required for the build phase.

The fundamental idea is based on a probabilistic analysis that examines the cardinalities of the attributes of the data set. Recall that an input set of size n is reduced to a view of size r because of the aggregation that takes place when records with identical attributes are encountered. Our goal therefore is to accurately estimate the number of duplicate records likely to be generated for a specific set of attributes. In the context of the data cube there are 2^d such sets, each corresponding to one cuboid. We note that this approach is similar to the one described in [40], though the implementation is somewhat different.

Theorem 3. *For an input set of size n, and a view V with a potential space of size S_v, we may estimate the number of records r in V by performing the summation*

$$x = \sum_{i=0} \frac{S_v}{(S_v - i)}$$

and terminating in one of two possible cases:

1. *$i >= S_v$, in which case $r = S_v$*
2. *$x >= n$, in which case $r = i$*

Proof. The estimation method is formulated as a counting problem in which we make n random selections from the potential space of V, and *replacement* is permitted (in order to represent duplicates). In each iteration i of the summation, we estimate the number of additional selections — $(S_v - i)/S_v$ — that would be necessary to obtain a non-duplicate, given that $i - 1$ unique values have already been found. To convert the raw probabilities into real numbers that may be used to represent the estimated number of additional selections required, we divide the result by one and add it to the running summation. The resulting equation is represented as:

$$x = \sum_{i=0} \frac{1}{(S_v - i)/S_v} = \sum_{i=0} \frac{S_v}{(S_v - i)}$$

To estimate the number of records required to construct V, we must provide the two terminating conditions.

1. $i >= S_v$. If this condition is satisfied, we know that the potential space has been fully saturated — implying that any additional selections from the input space will simply represent further aggregation. In such cases, we can estimate the size of V as S_v.
2. $x >= n$. In this case, we have used all n records without saturating the potential space. If this happens, then some degree of aggregation may have taken place and the number of *unique* records r in V formed by n random selections in the potential space can therefore be estimated as i.

□

We can use a simple example to illustrate our approach. Assume that five balls, each of a different colour, have been placed into a bag. We would like to determine the number of draws of a single ball required before three unique balls have been seen, given that the selected ball is returned to the bag after each retrieval. Using Theorem 3, the probability of obtaining a non-duplicate in each of the first three

iterations is $\frac{s-i}{s} = (5-0)/5 = 1$, $\frac{s-i}{s} = (5-1)/5 = 0.8$, and $\frac{s-i}{s} = (5-2)/5 = 0.6$. Incorporating these probabilities into the running summation, this would imply that we would require one selection to obtain one unique ball, approximately 2.25 selections to obtain two unique balls, and approximately 3.91 selections to find three unique balls.

In the context of the data cube, the expression can be interpreted as an estimate of the number of (unique) records found in a view V when drawn from a data set R. Since all records in V are unique (i.e., the objective of the cube operation is to aggregate duplicates into a single value), then the summation serves as a probabilistic estimate of the size of V. Because of its accuracy and efficiency, it is well suited to the costing model that we have developed for our PipeSort method.

3.6.2 Pipeline Cost Estimation

In the previous section, we described how the sizes of views in the lattice might be estimated. In this section, we describe a cost model that, given estimates of view sizes, estimates the time required to generate those views. The model takes into account the contribution of the following three fundamental processes:

1. Input/Output: One of the most expensive aspects of data cube generation is the writing of cuboids to disk. We must be able to capture this hardware-based cost, particularly in terms of its impact "relative" to RAM-based computation.

2. Pipeline Scanning: Each cuboid is constructed as the result of a scan through the sorted input set. Because the "scan" is shared by all views in a single pipeline, however, we must be able to describe the contribution of each individual cuboid.

3. Sorting: For each of the $\Omega(\binom{d}{\lceil d/2 \rceil})$ pipelines in the spanning tree, we must represent the costs of *either* Radix Sort or Quicksort on the appropriate input set.

When modelling the costs of the algorithm, we must strike a balance between *mapping granularity* and *model transparency*. In other words, our costing framework must be sufficiently detailed to capture the salient features of the algorithms, but not so detailed that it becomes unwieldy and difficult to modify. In the remainder of this section, we provide a complete description of the current data cube costing infrastructure.

3.6.2.1 Input/Output

The I/O metric is the one component of the cost model that is heavily influenced by the architecture of the machine being used. Specifically, I/O costs are intimately related to the hardware and supporting software upon which the data cube implementation actually runs. As such, it is necessary to experimentally define the relative impact of reading/writing data to disk. That being said, the impact cannot be defined in "absolute" terms; it must instead be a *relative* definition that can be measured against the scanning and sorting costs.

To represent I/O impact, we define what we call the *Write I/O Factor*. This expression represents the degree to which a "write" to the local physical disk unit is more expensive than a comparable write to a RAM buffer. In a generic application environment, such a measure would be virtually impossible to capture given the impact of random disk head movements and OS intervention. Recall, however, that our I/O manager assumes control of application input and output and reduces disk access to a finely coordinated sequence of streaming reads and writes. We can therefore accurately capture the I/O factor by experimentally comparing the cost of a RAM-to-RAM "write" versus a RAM-to-disk "write" for data sets of sufficient size (small writes can be dominated by the initial disk head movement). The Write I/O Factor for our local Linux cluster, for example, is 120. In other words, for streaming writes the cost of sending data to disk is a little more than two orders of magnitude

greater than that of accessing the same records in a memory buffer.

Since each individual record is composed of k feature attributes and a single measure value, the write cost for a k-dimensional data cube view with an *estimated* size of n is equivalent to $n(k+1) \times Write\ IO\ Factor$. The reader will note that this form of I/O metric can be deemed "relative" in that it is defined only in terms of n and k, parameters that will also be used by the sorting and scanning cost functions.

Finally, we note that there is actually a second I/O cost that must be captured in the cost model. In particular, views that are to be sorted must first be read into memory. "Read" I/O, however, is significantly less expensive than write I/O. For example, the *Read I/O Factor* on our current machine has been shown experimentally to be equal to 10, or approximately 1/12 of the Write I/O Factor.

3.6.2.2 Scanning

Though conceptually we think of scanning as representing the process of linearly passing through a parent view in order to compute a child, this is in fact not the case. Recall that the pipeline algorithm uses a single scan of the input set to compute all child views in the pipeline. In Section 3.5, we suggested that the scanning/aggregation phase could be bounded as $O(n * (k + m))$, where n was the number of records, k was the number of attributes, and m was the number of views in the pipeline. While this bound is valid, we cannot model scan costs in this fashion because during the spanning tree construction and partitioning phases, we do not actually know how many views will end up in a given pipeline. In other words, we do not know the value of m.

We deal with this problem by decomposing the scanning process into two components. First, there is the portion of the scan cost that is associated with passing through the input view and identify the i^{th} attribute in the set of k dimensions that represents an attribute change on contiguous records. We will in fact incorporate

this cost into the sort metric (discussed below). Second, there is the component of the scan that is associated with identifying a new record for view V (based upon the position of the attribute change), and moving that aggregated record to the I/O manager's output buffer. For a view V of size n, this cost is exactly equivalent to $n(k+1)$. Furthermore, this cost does not depend upon the number of views that end up in the final pipeline. Instead, each view must *always* be associated with this cost, regardless of its position in the pipeline or how it was created.

3.6.2.3 Sorting

Each pipeline is associated with a sort on some input view. We have described the process of dynamically choosing the appropriate sorting algorithm (Radix Sort or Quicksort). However, estimating the sort cost involves the computation of two other metrics. First, we must include the read cost of the input view, defined above as $n(k+1) \times Read\,I/O\,factor$. Second, we must capture the cost of finding the position of the first attribute that differs in contiguous records, the second component of the pipeline scan. While we cannot know where that position will be for an individual record (e.g, $1 \leq pos \leq k$), we can define the average or *amortized* position as $k/2$. In other words, we expect that, on average, we will have to look at half of the k attributes to find a change. The full cost of this portion of the scan is therefore $n * (k/2)$. We will refer to this cost as the *dim check* metric. Putting the three sort components together, we have $sort = read\,cost + (Radix \mid Quicksort) + dim\,check$.

We note that for the Standard pipeline implementation, costing of the Radix and Quicksort algorithms would be quite cumbersome due to the effect of data movement. Recall, however, that the use of vertical and horizontal indirection virtually eliminates such costs, allowing us to work with the much more manageable and more familiar $3kn$ and $n \log n$ metrics. Thus, the fully materialized sort metric is defined as:

$$(n(k+1) \times Read\ I/O\ Factor) + ((3kn) \mid (n \log n)) + (n \times (k/2))$$

As a final point, we must be careful to identify the appropriate input set for each cost metric. In the case of the sort, each of the three components is computed in terms of the existing view that must be resorted in order to compute the current pipeline. It is not in the pipe itself since it has already been created. The I/O and scan costs, however, are associated with views in the pipeline, the ones that are to be created from a scan of the sorted input set.

3.6.3 Putting it all together

While the previous sections have defined the cost functions that make up the larger model, the careful reader may have noticed that there is still one slight problem with the direct application of these metrics. Specifically, as described in Section 3.5, the I/O manager uses separate computation and I/O threads to exploit the functionality of modern DMA controllers. The result is that I/O and computation are to a large degree overlapped. We note that this does not mean that we can ignore I/O costs. Doing so would "suggest" to the bipartite matching algorithm that I/O costs were effectively zero — an assumption that is most definitely untrue — leading to a partitioning of work that would likely be poorly balanced.

At first glance, it would seem difficult to incorporate this knowledge into the model. Specifically, the I/O costs for a given partition are not known until the spanning tree has been cut; however, the cutting process itself relies upon accurate knowledge of the I/O costs. We deal with this problem by further augmenting the min-max partitioning algorithm. Recall that during each iteration of min-max, we evaluate the current cost of each partition in order to ascertain if the root partition is still the lightest. The algorithm generates these costs by summing the edge weights

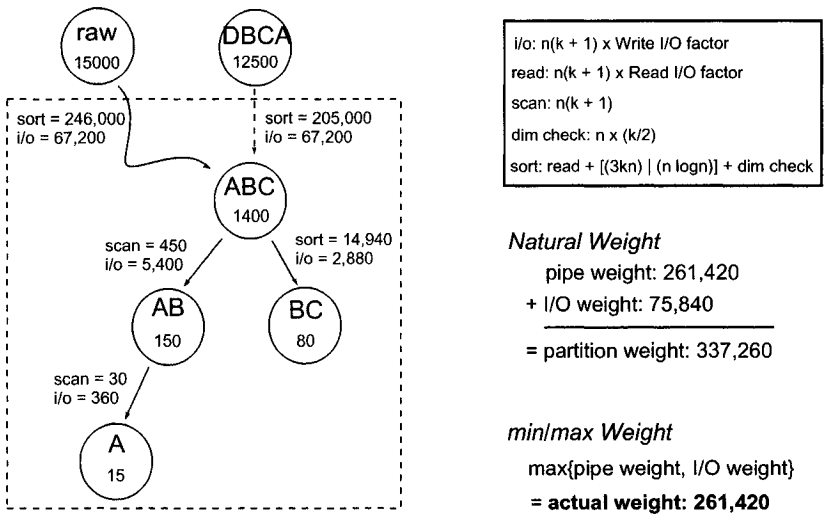


Figure 3.4: Dynamic cost calculations for a sample partition from a four dimensional space. The numeric values inside each view represent the estimated sizes.

of the current partitions, where the edge weights represent the costs of sorting, scanning and performing I/O. Let $\sum_{i=1}^{l} e_{full}$ denote the total cost for a partition with l edges, where all construction costs (i.e., sorting, scanning, and I/O) are serialized. To accurately represent the construction costs in the new multi-threaded design, the *pipe weight* $\sum_{i=1}^{l} e_{pipe}$ and the *I/O weight* $\sum_{i=1}^{l} e_{IO}$ are generated by decomposing $\sum_{i=1}^{l} e_{full}$ into its constituent components. We then define the *actual partition weight* as $max\{\sum_{i=1}^{l} e_{pipe}, \sum_{i=1}^{l} e_{IO}\}$. In other words, for the current partition we dynamically decide which of the two components, each with its own execution threads, will dominate or bound the cost of the current computation. Since the execution threads almost completely overlap, we know that the smaller of the two costs will be mostly "absorbed" by the larger cost and will have little effect upon final run-time.

Figure 3.4 provides an illustration of the complete model for a sample partition defined on a four dimensional space, including the cost metrics for each of the relevant computational components. The following costing features can be observed:

- First, note that the existence of the "parent" cut requires that the "raw" data

set be used as input for the initial sort.

- Edges from prefix-ordered pipelines are associated with scan and I/O costs, while non-prefix views are associated with sort and I/O costs.

- The *actual partition weight* in this example is bounded by the pipeline costs.

Finally, we note that the reader should not assume that the relative weights in this particular example are indicative of general trends. Each data cube problem is unique, and the size and nature of the data set, coupled with the number of feature attributes, will produce different partitioning patterns.

3.7 Implementation

In total, approximately 20,000 lines of code support the full data cube system, including the partial cube and indexing facilities described in subsequent chapters. Though some initial coding was done in C, we chose to move to a C++ platform in order to more efficiently support the growth of the project. With the expansion of the code base and the involvement of a number of independent developers, several of whom were in geographically distinct locations, it made more sense to employ an object-oriented language that allowed for data protection and class inheritance.

A number of third-party software libraries were also utilized. Node-to- node communication is supported by LAM's Message Passing Interface (MPI) [83]. Recall that efficient MPI implementations exist for both SMP and distributed memory architectures. Thread functionality is provided by the Pthreads (POSIX Threads) libraries [97]. Finally, we have also incorporated the LEDA graph libraries into our data cube code base [74]. We selected LEDA because of its rich collection of fundamental data structures (including linked lists, hash tables, arrays, and graphs), the extensive implementation of supporting algorithms, and the C++ code base [74]. Though there

is a slight learning curve associated with LEDA, the package has proven to be both efficient and reliable.

Having incorporated the LEDA libraries into our system, we were able to implement the lattice structure as a LEDA graph, thus allowing us to draw upon a large number of built-in graph support methods. In this case, we have sub-classed the graph template to permit the construction of algorithm-specific structures for node and edge objects. As such, a robust implementation base has been established; additional algorithms can be "plugged in" to the framework simply by sub-classing the lattice template and (a) over-riding or adding methods and (b) defining the new node and edge objects that should be used as template parameters.

The final system, though large, has been constructed to be as modular and extensible as possible. Extensions to the core algorithms may treat the current system as a data cube *back-end*, using its core methods as an API. Because of the emphasis on fundamental software engineering principles, the current data cube engine is more than a mere "proof of concept". It is a robust parallel OLAP engine that should continue to support related research projects for a number of years to come.

3.7.1 Generating Data Cube Input

In order to effectively evaluate the data cube algorithms, it is necessary to utilize a wide variety of test sets that reflect the patterns and idiosyncracies one is likely to encounter in practical settings. As such, we have designed our own *data generator* that produces user-defined integer-based records (non-integer records would typically be mapped to integers in an industrial application). The generator accepts parameters for such things as the number of records in the data set, the number of feature attributes, and the number of unique values in each dimension. Appendix D provides a detailed description of the data generation sub-system.

3.8 Analysis

Algorithm designers seek not only to produce elegant computational mechanisms, but also ones that demonstrate acceptable performance characteristics on working systems. In the sequential design world, this second theme is typically associated with some form of asymptotic analysis that attempts to bound the run-time of the algorithm in terms of its key parameters, most often the size of its input. In Appendix A, we note that the analysis of parallel algorithms tends to be more complex in that a number of additional parameters are involved. Analysis of message passing systems, in particular, must be able to incorporate the impact of slow communication fabrics. Models for the analysis of coarse grained parallel algorithms, such as BSP [121], CGM [33], and logP [23], might do this by bounding the computation on each node relative to that of the original input on a single processor *and* bounding the number of communication rounds required.

Given that one of the principle tenets of coarse grained design is the decomposition of the algorithm into a series of "supersteps", it should be clear that the current data cube algorithm fits this general design model quite well. Specifically, we reduce the number of such rounds to exactly one — task list distribution, followed by pipeline computation. Moreover, the communication in this superstep has total size of only $O(2^d)$ since it is used to pass each view in the spanning tree to one of the p processors. Subsequent to the communication phase of the superstep, each processor computes independently until the algorithm terminates.

Having said this, our parallel data cube algorithm does not lend itself to a standard coarse grained analysis. There are several reasons for this. First, the data cube *algorithm* is in fact a collection of sequential algorithms, making a concise (i.e., readable) analytic bound somewhat difficult. Second, the algorithm is a combination of in-memory algorithmic computation and large scale input/output phases. It is quite difficult to combine internal and external memory bounds within a single meaningful

expression. Finally, in order to provide asymptotic bounds on parallel computation, it is expected that one can say something precise about the balance of input partitioning across processors. Because weighted graph partitioning is NP-complete, and we must therefore rely upon heuristic solutions, it may not be possible to make such an assumption.

As a result, we must take a less direct approach to the analysis of our parallel data cube implementation. Specifically, we attempt to provide the following support:

1. Determine if the addition of the graph partitioning algorithm has altered the asymptotic bound of the scheduling phase.

2. Justify the use of the partitioning scheme, both logically and experimentally.

We will address these two issues in the remainder of this section. First, however, we make the following observation with respect to the algorithmic model. Occasionally, it is possible to design a parallel algorithm in which the workload is divided evenly between processors, which then independently compute their share of the workload. Such algorithms are often referred to as *embarrassingly parallel* to indicate the simplicity of their construction. The parallel data cube algorithm, despite its use of independent pipeline computation, is NOT such a design. Specifically, we exploit global information to make "up front" load balancing decisions rather than real-time local decisions, as is often done. In effect, by building an explicit cost model that takes into account the properties of the data and the underlying parallel machine, we simply migrate the load balancing logic from its traditional position within the run-time component of the algorithm to a distinct pre-processing phase.

3.8.1 The Scheduling Phase

We begin by examining the scheduling phase in Steps 1-4 of Algorithm 7. Here, we have taken the original sequential algorithm and extended it with a tree partitioning

algorithm. Clearly, run-time of the scheduling phase is now bounded as the concatenation of the sequential run-times for tree construction and tree partitioning. If the parallel algorithm is to be effective in practice, however, its running time (for typical parameter settings) should not be much larger than the run-time of the scheduling phase of the original algorithm. Otherwise, the increased cost of the sequential scheduling phase will make it difficult to obtain acceptable parallel speedup. More directly, we would like to know if the scheduling phase is bounded asymptotically by spanning tree construction or by tree partitioning?

Theorem 4. *For a lattice L of maximum width W, and a tree T extracted from L, the time complexity T_p of the tree partitioning algorithm dominates the time complexity T_c of the tree construction algorithm if, for p processors and d dimensions, $p(p^2d+2^d) > d^2W^2(d+\log dW)$.*

Proof. The time complexity of the tree construction algorithm is bounded by the d iterations (one for each level in the tree) of its bipartite matching algorithm. For a bipartite graph with n nodes and m edges this algorithm is $O(n*(m+n\log n))$ [80]. Recall that during each iteration of bipartite matching, we augment the original lattice by generating k copies of each view at the parent level, each of which is connected with an edge to k children at the child level. We note that while the number of nodes at a given level k can be represented by the combinatorial expression $\binom{d}{k} = \frac{d!}{(d-k)!\,k!}$, incorporating d such expressions into a single bound creates a very awkward result. Instead, we let W denote the maximum value of $\binom{d}{k}$ for $1 \leq k \leq d$. Simply put, W represents the widest level of the lattice. We will bound the algorithm as the cost of computing this one round of bipartite matching, rather than the cost of computing d rounds. While this understates the upper bound, it is perfectly acceptable in this context since our aim is to show that spanning tree partitioning is cheap relative to tree construction, even in this restricted form.

The number of nodes and edges for level W when it is augmented by the bipartite matching algorithm can now be written as $n = dW$ and $m = d^2W$, respectively. The final upper bound for tree construction can be expressed as $T_c = O(dW(d^2W + dW\log dW)) = O(d^2W^2(d+\log dW))$.

The second sequential algorithm, tree-partitioning, is bounded by the cost complexity of k-min-max [6]. The upper bound is $O(p^3H + pn)$ where p is the partition/processor count, H is the height of the tree, and n is the node count. With $H = d$ and $n = 2^d$, we have an bound of $T_p = O(p^3d+p2^d) = O(p(p^2d+2^d))$. We add that the "dynamic" extensions to k-min-max do not increase its asymptotic complexity. The use of both the "cut" hash table and overlap-based costing add only a constant number of operations to algorithmic steps that would be executed in the non-dynamic version.

	p = 4	p = 8	p = 16	p = 32	p = 64	p = 128
d = 4	C	C	P	P	P	P
d = 6	C	C	C	P	P	P
d = 8	C	C	C	C	C	P
d = 10	C	C	C	C	C	C
d = 12	C	C	C	C	C	C
d = 14	C	C	C	C	C	C

Table 3.1: Evaluation of scheduling phase for reasonable values of d and p. The dominant cost is listed as either C (spanning tree construction) or P (spanning tree partitioning).

Given the bounds of the two core sequential algorithms, we can conclude that the $O(p(p^2d + 2^d))$ min-max component will asymptotically dominate the $O(d^2W^2(d + \log dW))$ spanning tree component only when $p(p^2d + 2^d) > d^2W^2(d + \log dW)$. □

In Table 3.1, we analyze the inequality for typical values of p and d. Observe that the tree partitioning phase only dominates the construction phase when very high processor counts are combined with low dimension counts. However, this combination is unrealistic in practice since such small data cube problems simply do not require this degree of parallelism. Note as well that because we have understated the upper bound for tree construction, there may in practice be even fewer situations in which tree partitioning dominates. As such, we can say with some confidence that our parallel extensions to the scheduling phase are unlikely to distort the performance characteristics of the original scheduling algorithm for reasonable values of p and d.

3.8.2 Workload Partitioning

We judge coarse grained parallel algorithms not only on the quality of their sequential components, but upon their ability to effectively utilize available parallel resources. In other words, the goal is to equally distribute the workload across the p processors of our parallel machine. In Appendix B, we note that when the algorithm's workload is represented by a task graph, the resulting partitioning problem is often NP-complete. In such situations, we must resort to heuristic solutions which are expected to provide

"reasonably good" results in tractable time. This is the approach that we have taken with our parallel PipeSort.

Because provably optimal solutions are not available, however, it is important to provide reasoned support for the design decisions that have been made. In the remainder of this section, we discuss the rational behind the partitioning model that is at the heart of the Parallel PipeSort (Algorithm 6).

We first consider the following basic tree partitioning problem.

Problem 1. *Partition the spanning tree T into p sub-trees such that the size of the largest sub-tree is minimized.*

Note that an optimal solution to this problem is not defined as a *perfect* partitioning, where perfect indicates a partitioning in which the size of each sub-tree is identical. In fact, such a partitioning is unlikely to exist given the arbitrary weights of the individual edges.

In any case, given a solution to Problem 1, we could then consider giving each processor a single sub-tree for local execution. In the context of the PipeSort, minimizing the size of the largest partition would therefore correspond to minimizing the time taken on any single processor.

With respect to an algorithmic solution for Problem 1, note that an optimal partitioning of a weighted graph is in fact NP-complete. A trivial reduction from the NP-complete *Multiprocessor Scheduling problem* [46] provides the basis of this proof. We note that the Multiprocessor Scheduling problem consists of some number of time-limited (i.e., weighted) tasks that must be distributed across a multiprocessor system before the expiration of some fixed deadline.

In the case of the PipeSort, however, an ordering on the construction of views (i.e., children are constructed *after* parents) limits the distribution pattern of these views. The PipeSort partitioning problem can therefore be reduced from the processor scheduling problem known as *Precedence Constrained Scheduling*. While this related

problem is also NP-complete [46], there is in fact one special case of the general problem that has proven to be solvable in polynomial time. Specifically, if the task ordering takes the form of a spanning tree, then the partitioning problem belongs to the complexity class P. An optimal polynomial time solution for minimizing the size of the largest partition is therefore possible. In fact, the k-min-max algorithm provides this optimal solution.

However, while k-min-max produces an optimal partitioning of the schedule tree into p pieces, we note that the partitons/sub-trees may still be quite uneven in size. We therefore define a second problem in which the goal is to improve upon the basic partitioning provided by k-min-max.

Problem 2. *Partition the spanning tree T into β sub-trees and then group these sub-trees into p partitions (sets of sub-trees) such that the weight of the heaviest partition is minimized.*

Unfortunately, combining sub-trees in an optimal manner is also NP-complete. In fact, if we consider a sub-tree with j edges to represent a single task of weight $\Sigma_{i=1}^{j} weight(j)$, and we remove the precedence ordering, we can reduce the problem of optimal sub-tree combination to the original Multiprocessor Scheduling problem.

We address this new problem with the approximation technique described in Algorithm 6. This algorithm is an example of a *heuristic* approach to approximation. By this we mean that, while a provably optimal algorithm may not be possible, we can use our knowledge of the problem space to provide a solution that is likely to be good in practice. Specifically, we note that while tree-partitioning in the general case can produce partitions of widely varying sizes, the situation for PipeSort spanning trees tends to be somewhat more predictable. Typically, the edge weights of the PipeSort schedule trees grow gradually smaller towards the leaf nodes/views. Because there are no radically disproportionate costs associated with single edges (as there might be in the general case), k-min-max partitioning tends to produce an initial set of $s \times p$ partitions that are quite well balanced, typically within 20% to 30% of the mean.

We can effectively exploit this feature of the problem using the "high-low pairing" technique in Steps 4-10 of Algorithm 6. In fact, if we restrict our attention to the case where the over sampling factor is two — a very important practical case — we can show the following.

Proposition 3. *For an over-sampling factor of two and a partitioning where* $|subTree_{max}| \leq 2|subTree_{min}|$*, Algorithm 6 produces an optimal combination of sub-problems.*

Proof. The proof has two components. First, we show that there is no strategy for combining sub-trees in groups of two that is better than high-low pairing. Then we show that if we allow combinations of more or less than two sub-trees, we still do not get a better result.

We look first at the case where every tree consists of exactly two sub-trees. Note that an optimal pairing of two trees per node is equivalent to minimizing the size of the largest two-tree combination. Now, assume that we have used the high-low approach to partition the $2 \times p$ sub-trees into p pairs, each with a high and low element. For $1 \leq i < j \leq p$, we have $high_i \geq high_j$ and $low_i \leq low_j$. One of these pairings represents the largest partition. We will denote the position of this partition as k, where $1 \leq k \leq p$. To reduce the size of the largest partition, we must move the low_k value to a new partition. If we move low_k to a new position l, such that $l > k$, observe that because $low_l \geq low_k$, we will create a new partition at k that is at least as large as the original. Conversely, we can move low_k to a position l, where $l < k$. This *will* create a smaller partition at k. However, in doing so, we create a partition at l that is at least as large as the original at k since $high_l \geq high_k$. We therefore conclude that it is not possible to produce better two-tree pairings than those produced by the high-low method.

We now look at combination strategies that allow arbitrary numbers of sub-trees per partition. We note that if high-low partitioning has not been used to combine two sub-trees on each processor, then there must be at least one processor that receives only one sub-tree and at least one processor that receives three or more. Now, even if the three smallest sub-trees are placed into one partition and the largest single sub-tree is placed in another, we know that the difference in size between the largest and smallest partition can be no less than the difference between the size of this three-tree partition and the one containing only the largest sub-tree. Given that the largest sub-tree can be no more than twice the size of the smallest sub-tree, we can conclude that the difference between the largest and smallest partition is at least $3|subTree_{min}| - 2|subTree_{min}| = |subTree_{min}|$.

For high-low partitioning, we have a maximum size high-low pair at position k and a minimum size high-low pair at position l, $1 \leq k, l \leq p$. Let us assume first that $k < l$. Note that since both $high_l$ and low_l are at least as large as low_k, the difference between the sizes of the two partitions is maximized by setting $low_k = high_l = low_l$ and $high_k$ to twice the size of the three smaller sub-trees. We therefore have a maximum partition whose size is at most $2|subTree_{min}| + |subTree_{min}| = 3|subTree_{min}|$ and a

minimum partition whose size is at least $|subTree_{min}|+|subTree_{min}| = 2|subTree_{min}|$. Thus, the difference between the two is at most $|subTree_{min}|$. We now consider the case when $k > l$. Note here that since $high_l$ is at least as large as both $high_k$ and low_k, the difference between the sizes of the two partitions is maximized by setting $high_l = high_k = low_k$ and low_l to half the size of the three larger partitions. We therefore have a maximum partition whose size is at most $2|subTree_{min}| + 2|subTree_{min}| = 4|subTree_{min}|$ and a minimum partition whose size is at least $2|subTree_{min}| + |subTree_{min}| = 3|subTree_{min}|$. Thus, the difference between the two is again at most $|subTree_{min}|$. We can therefore conclude that the partitioning produced by high-low pairings is at least as good as that produced by arbitrary combinations of sub-trees on each processor.

Since both cases have now been proven, we have demonstrated that Algorithm 6 produces an optimal combination of sub-problems for an over-sampling factor of two and a partitioning where $|subTree_{max}| \leq 2|subTree_{min}|$.

□

As we will see in Section 3.9.4, application of the high/low technique significantly reduces the balancing error and is particularly effective when $sf = 2$.

3.9 Experimental Evaluation

In this section, we discuss the performance of our parallel PipeSort implementation under a variety of test scenarios. Evaluation was conducted on the two systems previously described in Chapter 1:

1. Linux Cluster. The Linux system (kernel version 2.4.9-13) is a 64-processor (1.8 GHz), 32-node Beowulf configuration, with a 100-Mb/sec switch supporting the interconnect. Each node has its own pair of 40 GB IDE disks. (We note, however, that to simplify the interpretation of results, only one CPU and one disk were utilized on each node.)

2. SunFire 6800 Cluster. The SunFire 6800 is a very recent SUN multiprocessor that runs the Solaris 8 operating system. It uses SUN 900 MHz UltraSPARC III processors with one GB of RAM per CPU. Finally, the SunFire comes equipped with and a SUN T3 shared disk array.

The majority of our testing is conducted on the Linux cluster. This is largely due to the limited access we had to a shared disk machine. Data cube evaluation is an extremely time consuming process. Not only do a very large number of individual tests have to be run, but the tests themselves can be very long running. Moreover, direct access to the file system is often useful. Since available time on the more expensive multiprocessor was quite limited, and because low level access to resources was not possible, the use of our own Linux cluster allowed for a much more complete suite of tests than would otherwise be possible. Having said that, it was imperative that we confirm the applicability of the algorithm to shared disk architectures as well as fully distributed clusters. As such, parallel Speedup and Efficiency graphs are provided for both physical targets.

We will look at a sequence of data cube tests, each designed to highlight one important characteristic or parameter of the model. In effect, we utilize a set of base parameters and then vary exactly one of these parameters in each of the tests. These base parameters are (with defaults listed in parenthesis):

1. Processor Count (16)
2. Fact Table Size (2 million rows)
3. Dimension Count (10)
4. Over-sampling Factor (2)
5. Skew (uniform distribution)

When default values are not, or cannot, be used for a particular test, this fact will be clearly noted. Finally, we add that each point in the graphs represents the mean value of three separate test runs. Note that in general the variation between runs was less then 3%.

3.9.1 Parallel Speedup

The most fundamental, and arguably the most important, of all parallel computing tests is the Speedup/Efficiency evaluation. Simply put, it is the primary measure of the value or usefulness of the parallel algorithm or system. Not only does it provide a snapshot of the quality of the workload balance, but it allows us to understand how the potential sources of overhead can affect run-time as we move beyond a single processor. It would have been nice to compare the speedup results produced by Algorithm 7 with the work of Goil and Choudhary [48, 49, 49]; however, they unfortunately do not give any speedup numbers.

Note that because we are using a coarse-grained parallel computing model, one that directly exploits existing sequential algorithms, it is possible for us to accurately compare parallel performance to that of efficient sequential techniques. Specifically, we run the the sequential component of the parallel code on a single processor to obtain accurate benchmarks.

The first Speedup test analyzes the performance of Algorithm 7 on the Linux cluster as processor count increases from 1 to 24. Figure 3.5 depicts the performance curve. Also shown is the optimal curve, calculated as $T_{opt} = T_{sequential}/p$. Note that the actual performance tracks the optimal curve closely. For 24 processors, our method achieves a Speedup of 20.18. In Figure 3.6, we provide the corresponding efficiency ratings. With smaller processor counts (i.e., $p \leq 12$), efficiency values lie somewhere between 90% - 95%, while for processor counts beyond this point, the ratings are between 83% and 90%. Given the complexity of the data cube problem, these results represent an extremely efficient use of parallel resources. Moreover, the relatively mild decrease in efficiency at 16+ processors suggests that even higher processor counts are likely to provide acceptable efficiency.

It is interesting to note that the efficiency curve has a slightly ragged shape. In other words, the curve from one to 24 processors is not strictly linear. In fact, this is

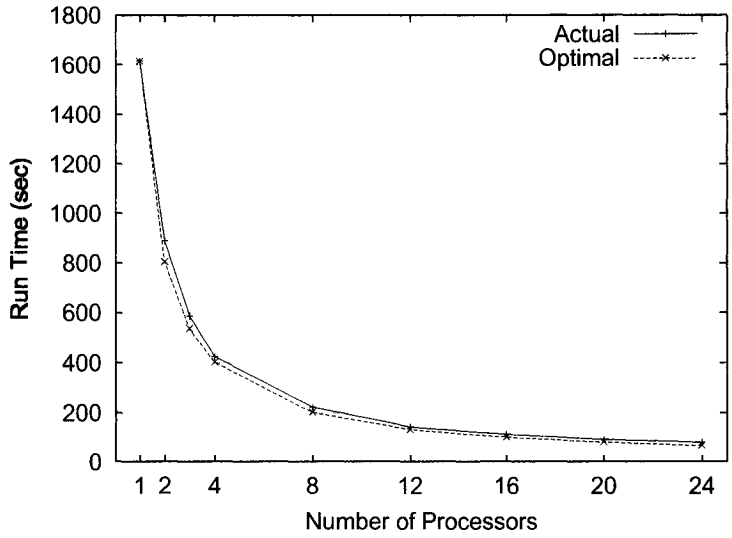


Figure 3.5: Speedup test for 1 to 24 processors on a Linux cluster.

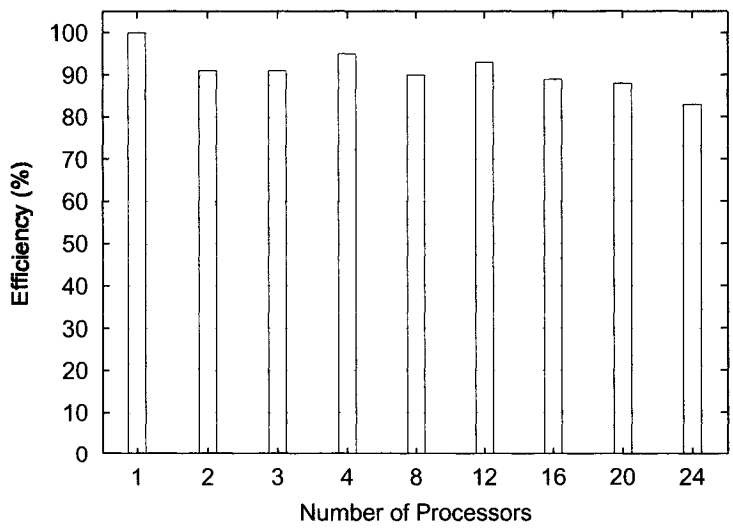


Figure 3.6: Efficiency ratings for the Linux cluster.

exactly as we would expect since the algorithm must provide a unique partitioning of the spanning tree for each processor count. Note that in this test, all evaluation runs use the same input set to ensure an "apples-to-apples" comparison. Consequently, it is possible for a larger processor count to get a slightly better partitioning than a smaller count (e.g., 12 versus 8).

In the second round of testing, we evaluate Speedup and Efficiency on the SunFire 6800 (note: only 16 processors were available for testing on the SunFire). Figure 3.7 shows the Speedup curve, while Figure 3.8 provides the efficiency ratings. From two to eight processors, the efficiency again exceeds 90%. At 16 processors, efficiency slips to about 80%. Since this is slightly more noticeable than the decrease on the Linux cluster, it begs the question, "Is this an algorithmic issue?" Our analysis suggests that the answer is likely no. Rather, it is the nature of the SunFire disk array itself that is beginning to affect performance. Specifically, the disk array houses a fixed number of independent disk units that may respond in parallel to I/O requests. This occurs for any number of application processes. In other words, even a sequential algorithm benefits from parallel I/O with a disk array. However, the fact that the number of disk units is fixed means that a "law of diminishing returns" eventually comes in to effect. By this we mean that on the SunFire, unlike the Linux cluster where the addition of a processor implies the addition of a disk unit, eventually the number of processors involved in the computation will exceed the number of disks in the array. Now, if the application does a significant amount of I/O, then this I/O may eventually become a bottleneck because the parallel system's ability to reduce I/O costs is more limited than its ability to reduce computation costs. Simply put, at higher processor counts, the I/O costs for the data cube can no longer be completely hidden by the pipeline computation costs.

These results, then, would suggest that shared disk data cube implementations would be best suited to parallel settings with a small to moderate processor count.

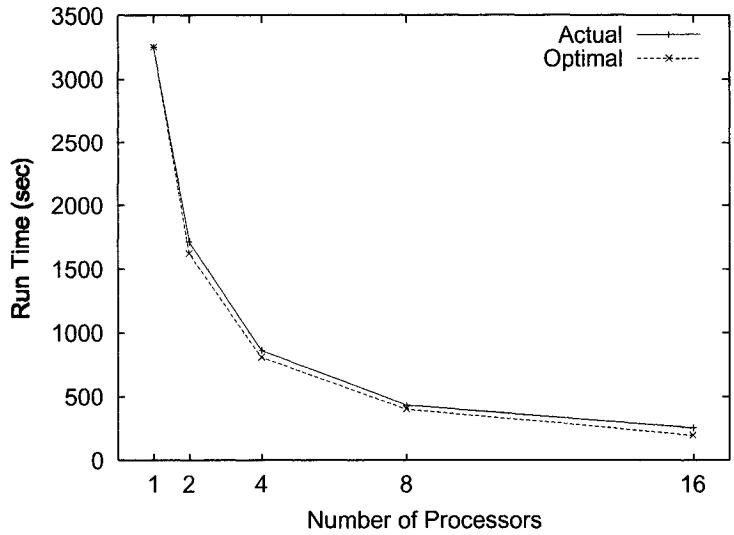


Figure 3.7: Speedup test for one to 16 processors on the SunFire 6800.

We note that, in practice, this observation would not likely be seen as a significant restriction. Specifically, because data warehouses would almost certainly be constructed as dedicated systems, unlike the multi-purpose MPPs often seen in scientific computing environments, small to moderate parallel systems would often be the platform of choice.

3.9.2 Data Set Size

In this section, we analyze the effect on performance as we increase the size of the input set, holding other parameters constant. Figure 3.9 depicts the running time on the Linux cluster for data sets ranging in size from 500,000 records to 10,000,000 records. Before examining the curve, we note the following. As we increase problem size, the performance of the I/O component of the algorithm will effectively increase at a linear rate since it essentially performs streaming writes to disk. On the computational side, however, run-time is bounded by the time to perform sorting. Recall that we may either use QuickSort of Radix Sort for this purpose. In higher dimensional space (e.g., the 10-dimensional set used for the test), Quicksort will generally be chosen for

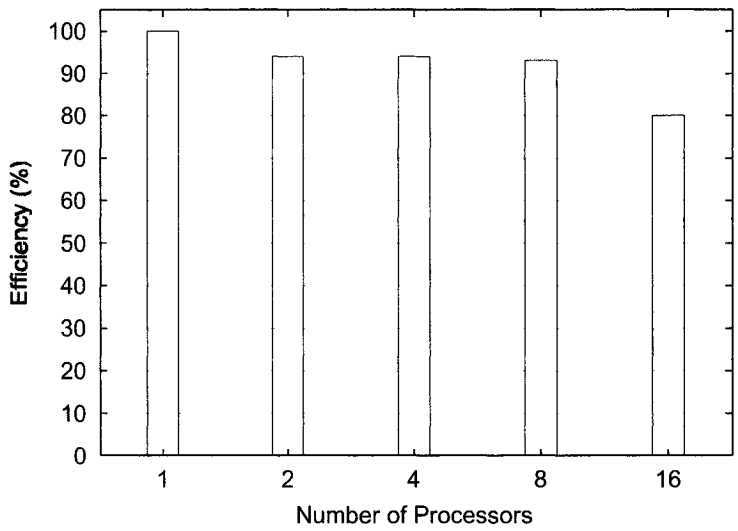


Figure 3.8: Efficiency ratings for the SunFire 6800.

smaller data sets, but will give way to Radix Sort once the sets become very large. Thus, we would expect to see a slightly super-linear growth pattern for the smaller sets (due to the $O(n \log n)$ Quicksort), but then a more or less linear pattern after a certain point (due to the asymptotically linear bound of Radix Sort).

Returning to Figure 3.9, we see a curve that closely resembles the expected shape. On small sets, there is a modest super-linear increase in run-time. Between 5 million and 10 million records, however, we see a curve that is almost perfectly linear in shape. At this point, almost all of the sorting is performed by Radix Sort. Consequently, the curve begins to flatten out. This, in fact, is a very encouraging sign for data cube implementations that will deal with even larger fact tables.

3.9.3 Dimension Count

In this test, we examine the effect on increasing the number of dimensions for data sets with a fixed row count (the default of 2,000,000). Figure 3.10 illustrates the effect of increasing the number of dimensions in the problem space from six to 14. Recall that an increase of one in the dimension count corresponds to a doubling in

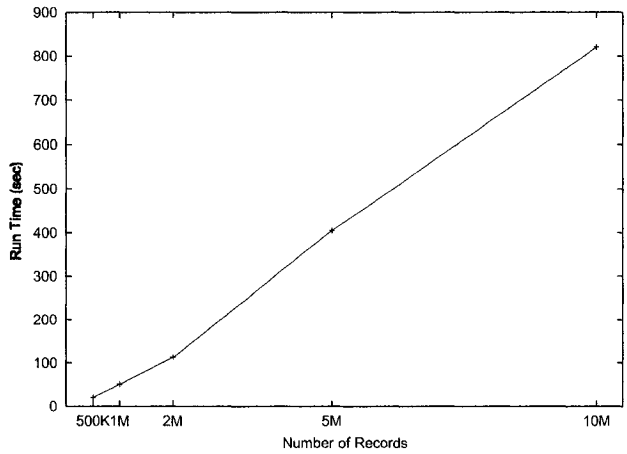


Figure 3.9: Record count evaluation.

the number of views to be produced. Interestingly, the general shape of the curve is quite similar to that of the Record Count test. The explanation for this, however, is slightly different. In this case, the combination of data set size and dimension count is likely to favour the QuickSort for most of the large sorts. It is important, incidentally, to distinguish between large and small sorts because even though most of the views in the lattice would have smaller dimension counts and would thus be sorted with Radix Sort, it is the large views at the top of the pipelines that dominate overall sorting costs. In any case, based upon this observation one might expect this curve to continue to increase at a super-linear rate. It does not do this because I/O costs become increasingly important as we pass 10 dimensions. This is the case because in high dimensional space a large percentage of the views are almost as large as the original fact table. Given the the significantly penalty of writing all of this data to slow secondary storage, the I/O phase of the algorithm begins to dominate the in-memory computation. Since the I/O in this system is strictly linear (i.e., it is streaming, not random access), the growth curve again begins to flatten out. We may therefore conclude that for significantly large problems, either in terms of data set size or dimension count, increases in input parameters result in a roughly linear increase in run-time performance.

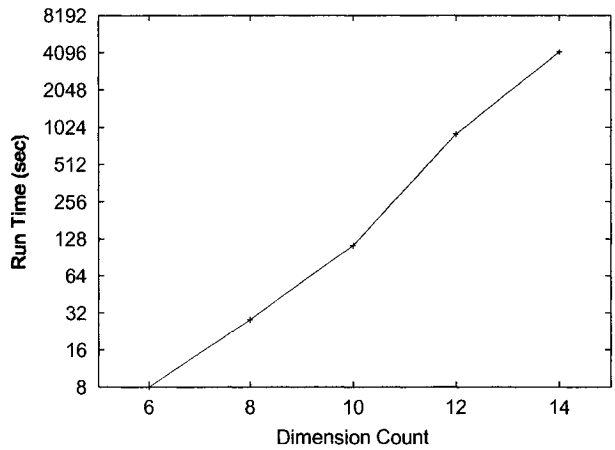


Figure 3.10: Dimension test

3.9.4 Over-Sampling Factor

Recall that the use of a sample factor represents an attempt to reduce run-time by improving the quality of workload partitioning. Figure 3.11 provides experimental evidence that our approach is justified. In both test cases, we vary the sampling factor *sf* from one (i.e., no over-sampling) to four. Figure 3.11(a) presents results for the default data set of two million records, while Figure 3.11(b) looks at 10 million records. The results are virtually identical. In both cases, the use of an over-sampling factor, with $sf = 2$, reduces the *base* run-time by approximately 35%. This reduction in run-time is entirely the result of improved partitioning. For larger values of *sf*, however, performance actually begins to decline. There are two reasons for this. First, as previously noted, each additional partition on each compute node is associated with a slightly more expensive sort of the raw data set. Second, as we continue to partition the original tree, we create sub-trees with increasingly shorter pipelines. The effect is to reduce the degree to which sorts can be shared by multiple views. As such, it appears quite likely that an over-sampling factor of two will be optimal for virtually all practical parallel architectures.

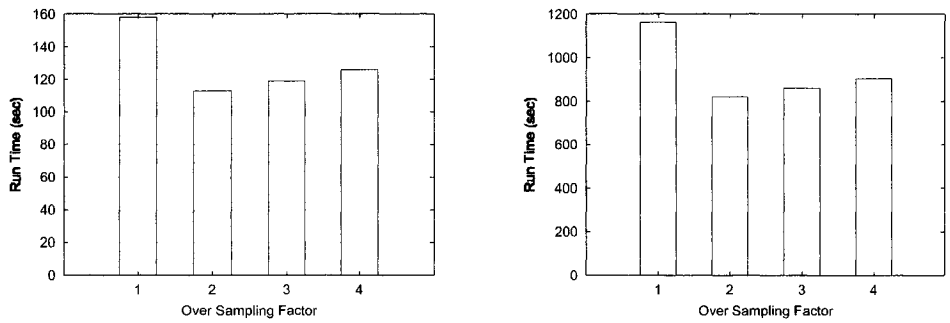


Figure 3.11: (a) Sample factor: two million records. (b) Sample factor: 10 million records.

3.9.5 Record Skew

In this section we present tests that are designed to illustrate the impact of record skew on the algorithm's costing model. Recall that the current size estimator assumes a uniform distribution of data. One would expect then that as the values of each dimension become increasingly skewed (i.e., many more occurrences of some dimension values than others), the size estimates would become increasingly inaccurate. By extension, load balancing decisions would suffer, resulting in larger run-times. Figure 3.12 presents performance results for data sets that have been created with varying degrees of skew. We note that skew is produced with a *zipfian* function [123], a technique commonly employed in the data cube literature [110, 10] (see Appendix D for a description of the estimator application). In this case, $zipf = 0$ corresponds to no skew, $zipf = 0.5$ to moderate skew, and $zipf = 1$ to heavy skew. As the graph demonstrates, however, there is relatively little difference between the results for uniformly distributed data and those of the skewed sets.

In fact, there are two reasons for this. First, though estimation quality does decrease, resulting in sub-optimal load balancing, this degradation is partly offset by reductions in I/O costs for skewed data. Specifically, the introduction of significant skew creates more clustered data cube output, thereby reducing view size and, in

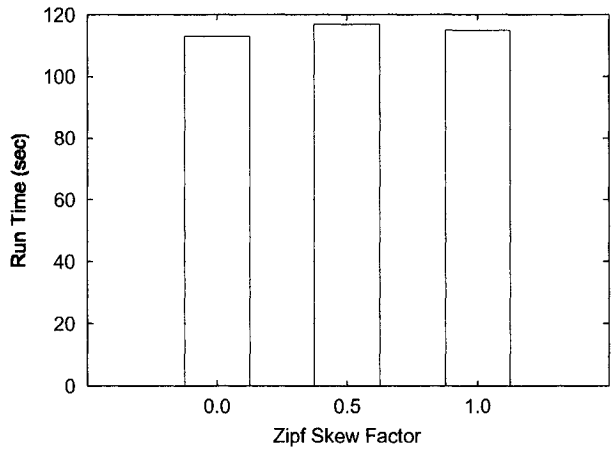
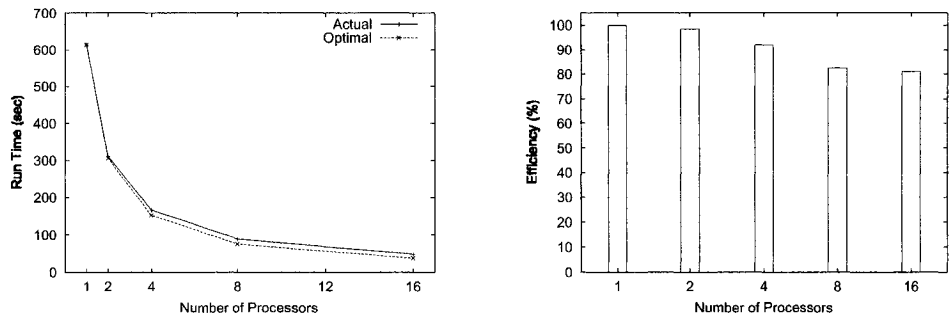


Figure 3.12: Skew test

turn, reducing run-time. Second, the use of the zipf function for data cube input sets tends to create a very regular pattern of skew. In particular, all dimensions are skewed to the same degree, with the result that the associated errors in size estimation tend to be "amortized" across the network. In other words, costing errors are unlikely to be disproportionately associated with any one node. The result, then, at least with this form of synthetic skew pattern, is that the algorithm handles estimation errors somewhat better than one might expect.

Having said that, it is clear that not all data sets would exhibit this kind of regularity in their skew patterns. Specifically, real world data sets are much more likely to have arbitrary clusters and pockets of skew that might lead to more irregular workload partitioning. We have therefore obtained a one million record data set containing information on weather patterns. While not a typical OLAP subject, the weather set makes an appropriate test set in that it consists of categorical attributes of reasonably low cardinality, and it has a meaningful "total" field that can serve as the measure attribute. In our case we have extracted the following 10 feature attributes from the 20-attribute records (cardinality in parenthesis): hour of day (24), brightness (2), present weather (101), lower cloud amount (10), lower cloud base height (11), low cloud type (12), middle cloud type (13), high cloud type (11), solar altitude (1800),



(a) (b)

Figure 3.13: (a) Speedup: 1 to 16 processors. (b) Corresponding parallel efficiency.

and relative illuminance (218). The measure attribute is total cloud cover (9).

Figure 3.13(a) depicts the parallel speedup on the weather data set for one to 16 processors (both actual and optimal), while Figure 3.13(b) provides the efficiency ratings. Observe that as the processor count increases to 16 processors, there is a slightly more noticeable decline in efficiency than was seen in the original cluster testing in Section 3.9.1. We believe this to be the result of estimation error than is not evenly distributed to each of the p nodes. Nevertheless, given that the current size estimator is not designed to handle this situation, parallel efficiency still exceeds 80% at 16 processors. This suggest that a *skew conscious* estimator might provide resource utilization similar to that obtained on uniformly distributed data sets.

3.9.6 Pipeline Performance

In this final test, we examine the performance benefits produced by the inclusion of the optimizations described in Section 3.5. Since we are specifically dealing with the sequential algorithm in this case, we provide performance results on a single processor. In particular, we contrast the run-time of the pipeline construction phase for the Standard algorithm with that of the Optimized Algorithm. We note that the Standard algorithm represents the high level description of the PipeSort, as presented in [105] and consists of:

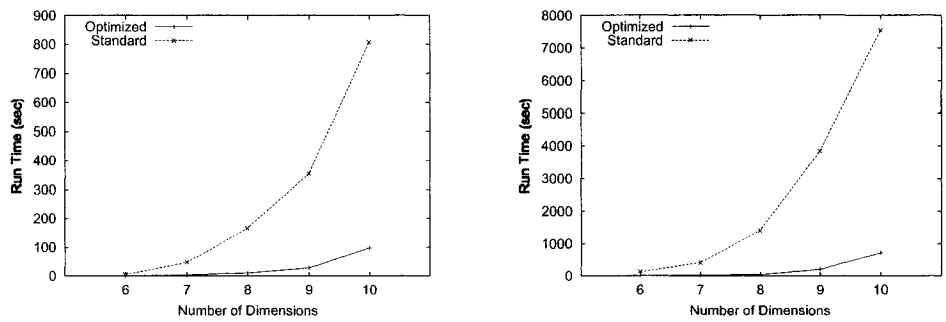


Figure 3.14: (a) Performance comparison for sequential PipeSort on 10^5 records. (b) The same comparison for 10^6 records.

- Quicksort-based sorting of all input sets
- Direct record-oriented operation (i.e., no use of vertical or horizontal indirection)
- *Naive* aggregation policies
- Operating system controlled I/O

Figure 3.14(a) contrasts the performance of the two sequential algorithms on 100,000 records, while Figure 3.14(b) looks at a data set of $1,000,000$ records. The curves are very similar in both cases and clearly demonstrate the runtime reductions achieved with the Optimized algorithm. At ten dimensions, for example, the runtime for the Standard algorithm is approximately 1000% greater than that of the Optimized algorithm (808 seconds versus 98.4 seconds on 10^5 records, and 7548 seconds versus 724 seconds on 10^6 records). Another way of saying this is that a single processor machine running the new algorithm (assuming adequate resources) would likely beat a 10-processor parallel machine running the original code.

3.10 Review of Research Objectives

In Section 3.3, we identified a number of objectives for this phase of the research. We now review those goals to confirm that they have in fact been accomplished.

1. **Build upon proven, optimized sequential algorithms for local computation.** In our case, we incorporate the sequential PipeSort algorithm into our parallel design. The PipeSort is one of the best known top-down methods for computing the data cube.

2. **Exploit well studied problems in the parallel computing literature for the purposes of workload distribution.** We represent the data cube workload as a task graph. Though general graph partitioning is NP-complete, a k-min-max algorithm, coupled with a technique for over-sampling, provides impressive load balancing.

3. **Minimize the communication costs due to view re-distribution.** A subset of views is computed locally on each node. As such, once the task lists are distributed, no further communication is required.

4. **Employ global costing information to ensure that local computation is partitioned/balanced as equitably as possible.** Partitioning is performed "up-front" on the full task graph. All costing information is therefore available when partitioning decisions are made.

5. **Support straightforward integration with standard relational systems.** The algorithm is an example of a ROLAP data cube method, meaning that its input and output are standard relational tables.

6. **Minimize the complexity — data structures, algorithms, resource management — that would likely lead to an unworkable practical implementation.** The algorithmic model is structured around "streaming" I/O and sorting/scanning of contiguous records, and thus avoids page-based data access. Efficient memory and I/O management is further streamlined by a relatively simple I/O manager that is embedded directly into the PipeSort

framework.

3.11 Conclusions

In this chapter, we have described a new method for the parallelization of the data cube, a fundamental component of contemporary OLAP systems. We have described an approach that models computation as a task graph, then uses graph partitioning algorithms to distribute portions of the task graph to each node. Locally, a view generation algorithm efficiently computes its portion of the workload.

As noted in Section 3.1, relatively little work in the area of parallel data cube construction has been published. Given the significance and size of the underlying problem, there would appear to be a genuine need for this type of research. In our case, we have contributed to the literature by providing one of the most comprehensive frameworks for the computation of the full cube. Of particular significance is the fact that we have grounded our algorithmic work with extensive experimental evaluation on commonly used parallel machines. We have shown that our approach is load balanced, communication efficient, viable in higher dimensions, and capable of being seamlessly integrated into standard relational database management systems.

Chapter 4

Computing Partial Cubes in Parallel

4.1 Introduction

In Chapter 3, we described a parallel algorithm for the generation of the full or complete data cube. Recall that the full data cube is one in which all 2^d possible cuboids are physically materialized. In this chapter, we describe a new algorithm for generating partial cubes, that is, data cubes in which a given subset of views are materialized. Partial cubes are of particular interest when the number of dimensions and/or the number of records is high. In such cases, the fully materialized data cube would be hundreds, if not thousands, of times larger than the original fact table, making it infeasible in terms of both compute time and storage requirements. Moreover, there are some settings in which users simply don't require all of the materialized cuboids. This would be the case, for example, when the primary role of the data cube was to support OLAP visualization functions. In such environments it would clearly be difficult to represent or even interpret high dimensional views in the upper part of the lattice. For both of these reasons, then, it is imperative that algorithms for the construction of view subsets be developed.

Perhaps surprisingly, very little research has been published on the generation of partial cubes. The few approaches that have been suggested [49, 10, 105] are either

infeasible, inapplicable, or limited in scope. In this chapter, we present a new *greedy* approach that can be used to efficiently generate scheduling trees for the construction of partial cubes. In combination with the parallel partitioning model discussed in the previous chapter, the new approach leads to efficient algorithms for both single and multi-processor architectures.

The chapter is organized as follows. In Section 4.2, we examine the previous work on partial cubes. The motivation for the new research is briefly summarized in Section 4.3, with the algorithmic foundation of the new partial cube framework presented in Section 4.4. In Section 4.5, we investigate performance and optimality issues, and present heuristic extensions that permit the algorithm to be pushed into high dimension spaces. Section 4.6 describes the method for parallelizing the partial cube method. Experimental results are highlighted in Section 4.7. Section 4.8 revisits the list of objectives identified in Section 4.3, while Section 4.9 offers some final observations.

4.2 Related Work

As noted in the introduction, there has only been a couple of results regarding partial cubes reported upon in the literature. In this section we examine the work that has been presented and explore in some detail a proposal for a PipeSort-oriented partial cube algorithm suggested by the authors of the original PipeSort paper. We begin, however, with partial cube solutions for two of the competing data cube algorithms presented in previous chapters.

In Chapter 2, we described a *bottom up* algorithm known as BUC [10] that was specifically tailored to large sparse spaces. Recall that BUC proceeded from low dimensional cuboids to high dimensional cuboids, recursively subdividing the current partition to sort/aggregate on increasingly smaller, finer granularity partitions. In so doing, BUC was able to minimize the size and cost of intermediate sorting operations

and thereby reduce its final run time. As a second benefit, the authors of BUC suggested that by terminating the recursion once the algorithm reached a specified granularity (i.e., number of attributes), they could in fact materialize only a portion of all cuboids. In other words, BUC can be adapted to compute a partial cube.

There are, however, two significant problems with the use of BUC as a partial cube algorithm. First, generating partial cubes by terminating the recursion effectively produces a computational "fence". In other words the algorithm produces a logical line that divides the lattice into two pieces. All cuboids below the line are computed, while all cuboids above the line are not. Though appropriate for some partial cube problems, the solution is clearly quite limited since it is not possible to specify an arbitrary subset of views that would be most appropriate for a particular environment.

Even when the general solution is appropriate — when low dimensional visualization is the primary goal, for example — a second problem arises. Specifically, the authors of BUC explicitly acknowledge that their algorithm performs most effectively in sparse environments requiring relatively little aggregation. For cuboids with fewer dimensions (say, five or less), the BUC algorithm performs poorly and is likely to be significantly outperformed by a number of competing solutions, including PipeSort. We can therefore conclude that while the algorithm can be adapted to produce partial cubes in the lower levels of the lattice, it is not particularly well suited to this task.

A second partial cube proposal was suggested by Goil and Choudhary [49]. Recall from Chapter 3 that this research focused on parallel MOLAP computation. Nevertheless, the authors also used a bipartite matching algorithm to determine the manner in which cuboids would be computed. Recognizing the obvious importance of partial cube computation, they suggested that their bipartite matching technique could be replaced with a second algorithm that builds partial cube scheduling trees as follows. Parent-child pairings would be established by eliminating unnecessary intermediate views from the tree. This would be accomplished during a bottom-up, level-by-level

traversal of the graph, in which only those views from level $k+1$ required to build views at level k would be included in the tree. This iterative process would continue until the base cuboid was reached.

Unfortunately, while this method does produce a partial cube schedule tree, the quality of that tree is in some doubt. The problem is that no attempt is made to select the "most appropriate" parent views at level $k+1$. Whereas a bipartite matching algorithm was used in the original model to find the cheapest means by which to construct the required set of child views, the new algorithm suggests that we make a linear scan of the views at level $k+1$, simply eliminating potential parents whose children can be computed from a previously included parent. While this approach does indeed eliminate unnecessary intermediate views, no attempt has been made to find the parent set that most reduces scheduling costs. The authors acknowledge that such a cost driven method would be desirable, but provide no means by which to accomplish this.

A second important observation about the partial cube approach in [49] is that the notion of automatically including *non-essential* parent views at level $k+1$ solely in order to compute views at level k is in fact fundamentally flawed. By "non-essential", we mean views that have not been identified by the user as necessary, but whose inclusion might reduce the construction cost of the selected set. We note that while a parent v at level $k+1$ may have a number of *potential* children at level k, we do not know in advance which, if any, of these children will actually use v as a parent. Quite often, such a parent might only be useful for the computation of a single child in the final version of the schedule tree. In such cases, it may actually be more expensive to compute a child from a parent at level $k+1$ than from a potentially larger parent at level $k+2$. Figure 4.1 provides a simple illustration. Here, the algorithm automatically includes the non-essential view ABC, the logic being that it will be used to more efficiently compute AB. However, in this case, AB is the

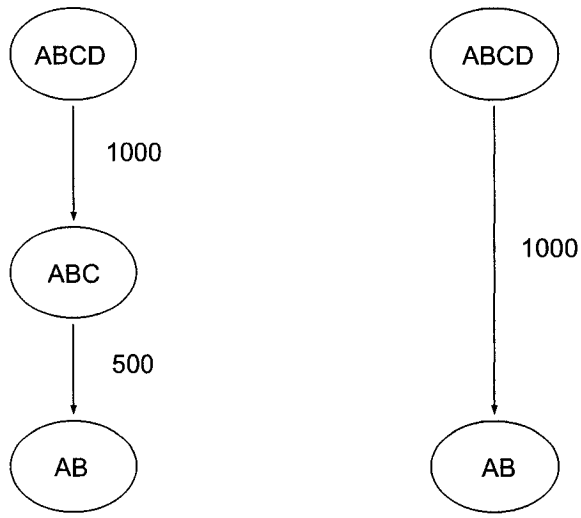


Figure 4.1: Two options for construction of the "essential" view AB.

only child of ABC and we are now effectively processing two parents in order to generate a single child. This type of *blind* inclusion has the potential to seriously inflate scheduling costs in partial cube settings. As such, it is unlikely that the Goil and Choudhary model would be an effective option for partial cube generation. In a parallel environment, in particular, much of the benefit provided by an increase in resources would be offset by poor scheduling.

Finally, a third approach to partial cube generation was provided by the authors of the original PipeSort algorithm [105]. Recall that for full cube computation, bipartite matching was used to select the sort/scan edges between successive levels of the lattice. Since the schedule tree for a partial cube may require edges between nodes at arbitrary levels of the lattice the authors suggest augmenting the lattice with *Steiner* vertices and edges. We note that in this context a "Steiner" node (or edge) is simply one that is not part of the original lattice. Because we are not able to set the sort paths using the level-by-level progression, we must allow for all possible paths in the Steiner representation. We therefore build a graph in which all possible permutations of each attribute ordering are represented. Furthermore, every permutation of every node

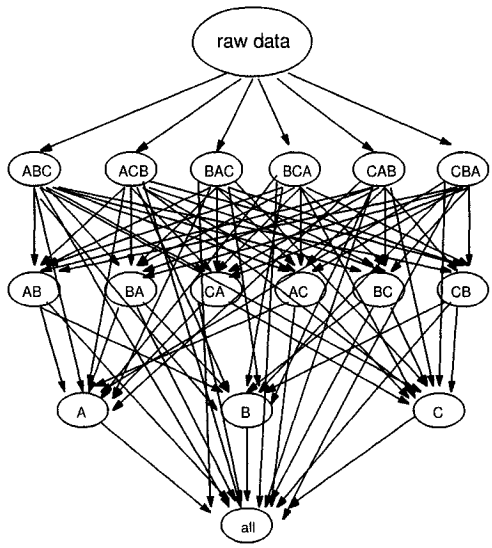


Figure 4.2: A three-dimensional lattice augmented so as to support a minimum Steiner tree algorithm.

must have an edge connecting it to every possible descendant. Figure 4.2 illustrates the graph for a "small" three-dimensional space.

The authors then apply a minimum Steiner tree approximation algorithm to the augmented lattice in order to create a schedule tree. The main problem with this approach, besides the fact that the minimum Steiner tree problem is NP-complete, is that the augmented lattice can become extraordinarily large. Equation 4.1, below, depicts the node count for the original lattice, while Equation 4.2 gives the edge count. Note that d is the number of dimensions in the lattice and k is the number of dimensions in each view at a given level, $1 \leq k \leq d$. For the Steiner augmentation, Equation 4.3 provides the node count, while Equation 4.4 gives the number of edges.

$$\sum_{k=0}^{d} \binom{d}{k} \tag{4.1}$$

$$\sum_{k=1}^{d} \binom{d}{k} k \tag{4.2}$$

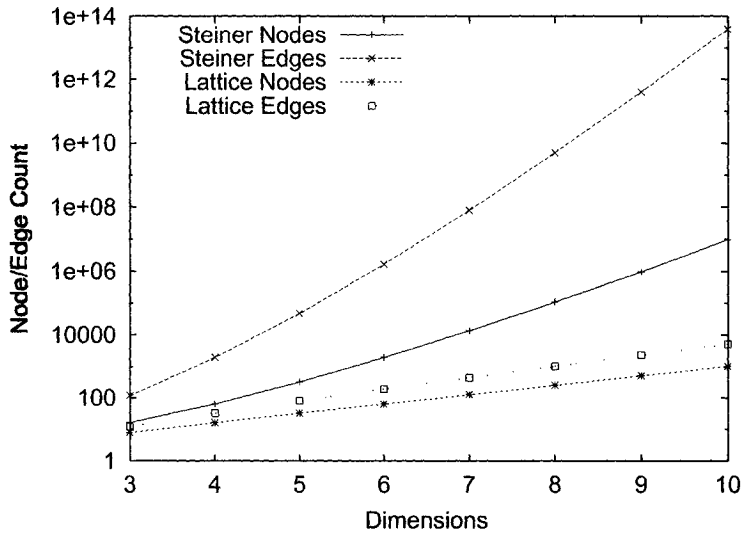


Figure 4.3: Growth patterns for Steiner graph versus original lattice. Note the logarithmic scale.

$$\sum_{k=0}^{d} \binom{d}{k} k! \tag{4.3}$$

$$\sum_{k=1}^{d} \left[\binom{d}{k} k! \sum_{j=1}^{k} \frac{k!}{(j-1)!} \right] \tag{4.4}$$

Figure 4.3 graphically illustrates the effect of applying the Steiner augmentations in three to ten dimensions. Table 4.1 provides the same information in tabular form. The number of Steiner edges is greater than 79,000,000 for $d = 7$ and reaches almost 40 trillion for $d = 10$. This makes such an approach impractical for data sets with more than just a very small number of dimensions. These examples suggest that, in order to handle high dimensional data sets, it is important to find approaches that do not require Steiner vertices and edges in the lattice.

Dimensions	Lattice Nodes	Lattice Edges	Steiner Nodes	Steiner Edges
3	8	12	16	117
4	16	32	65	1948
5	32	80	326	47665
6	64	192	1957	1667286
7	128	448	13700	79777285
8	256	1024	109601	5013145600
9	512	2304	986000	400328720384
10	1024	5120	9864101	39581776871424

Table 4.1: Growth patterns for Steiner graph versus original lattice in tabular form.

4.3 Motivation

While the computation of the full data cube has attracted considerable attention within the research community, it is nonetheless true that full cube generation for dimension counts above seven or eight is prohibitively expensive in many real world settings. Moreover, as noted in the preceding section, many of the views within a high dimension space may be of little practical benefit since high dimension cuboids may be difficult to visualize and interpret. Having said this, many data warehouses *do* consist of more than seven or eight attributes. For data cube algorithms to be relevant in practice, we must therefore find ways of applying the principles and benefits of full cube algorithms to the partial cube problem.

In designing these new partial cube algorithms, we attempt to satisfy the following fundamental goals:

1. Compute an efficient scheduling tree for the set of cuboids selected by the user.

2. The schedule tree should include "non-essential" views if they reduce the global cost.

3. The algorithm should be computable/tractable in high dimensional spaces — for example, those of 12 or even 14 dimensions.

4. The algorithm must be amenable to parallelization.

While this list of requirements is relatively brief, it must be noted that none of the three partial cube algorithms described in Section 4.2 is able to satisfy all of them. At the conclusion of this chapter, we will return to this list and confirm that its objectives have been reached.

4.4 A New Partial Cube Method

In this section, we present the details of a new algorithmic model for the construction of efficient partial cube scheduling trees. We note that although we have identified a number of general objectives in Section 4.3, our initial focus will be upon the process of building the tree and its associated partial cube. In subsequent sections, we will return to the issues of performance and scalability.

Before proceeding with a presentation of the new algorithms, we first comment on the complexity of the underlying problem. At present, it is not known whether an optimal solution for PipeSort schedule tree construction — full or partial — is possible in polynomial time. In fact, the authors of the original PipeSort algorithm explicitly acknowledged that their full cube algorithm was not provably optimal since it could not consider the inclusion of edges that spanned multiple levels in the lattice. Moreover, their partial cube approach was based upon an approximation algorithm for the NP-complete Minimum Steiner Tree problem. We note as well that while its complexity remains an open issue, PipeSort spanning tree construction is an example of a graph optimization problem, a general class of problems noted for the number of NP-complete instances it contains. Indeed, the problem is quite similar to Multiple Choice Branching, another optimization problem already known to be NP-complete [46].

Given the strong possibility that an optimal polynomial time solution for PipeSort spanning tree construction does not exist, we can therefore conclude that an approach based upon the identification of an approximate solution is well justified. In the

remainder of this section, we present a suite of heuristic algorithms for the generation of partial cube schedule trees. Extensive experimental support for the choices we have made will be provided in Section 4.7.

4.4.1 Adding Non-Essential Nodes to the Selected Set

We begin by reiterating that a final scheduling tree T should contain not only the views of the selected set S, but any additional nodes from the lattice L that might lower the global cost of T. This is in fact more than a semantic distinction. Specifically, there is no implicit guarantee that the logic used to add a node g from S is, or should be, the same as that used to add a *non-essential* node h from the set $L - S$. As such, we will find it advantageous to model the partial cube algorithm as a *pair* of cooperating algorithms, one for building an *essential* spanning tree E — consisting of the nodes found in S — and the other for adding non-essential nodes to E to produce a *reduced tree* R.

It is the addition of non-essential nodes that we will discuss first. In other words, we will for the time being assume that all of the nodes in S have already been arranged in a spanning tree E such that the cost of producing each node has been minimized. As such, our current objective is to examine the list of non-essential nodes — taken from the original lattice L — in order to determine if it is possible to add them to E, producing a new tree R, where $cost(R) < cost(E)$.

To accomplish this goal, we will utilize a *greedy algorithm*. Greedy approaches are common to optimization problems [22]. Here, at each decision point, the algorithm proceeds by making a locally optimal selection; it does not explicitly consider the global state of the system. In some cases, greedy algorithms can be shown to actually produce globally optimal solutions. Such is the case, for example, with Dijkstra's Single Source Shortest Path algorithm [22]. When the solution is not provably optimal, it is often possible to demonstrate experimentally that the greedy algorithm produces

solutions that are good in practice.

Our new greedy method for adding non-essential nodes to a schedule tree in order to reduce its total cost is described in Algorithm 10, AddNonEssentialViews. The algorithm builds upon the concept of "plan" variables, objects that maintain information about the costs of adding a given node to the current version of R. More precisely, a *plan* variable contains the following fields: (1) node: the node v being considered for insertion, (2) parent: the chosen parent of v, (3) parent mode: the chosen construction *mode* (scan or sort) for computing v from its parent, (4) scan child: the chosen child of v that is computed via a scan, (5) insertion scan child: the chosen scan child of v in the case of scan insertion (the term "scan insertion" will be explained shortly), (6) sort children: the chosen children of v that are computed via sort, (7) benefit: the improvement in total cost obtained by inserting v.

Algorithm 10 Add Non Essential Views

Input: A tree E consisting of the selected group-bys, and a guiding graph G. Also used are auxiliary variables BP (best plan) and CP (current plan).
Output: Reduced tree R.

{Add nodes from $G - R$ to E as long as the total cost improves}

```
1: repeat
2:   clear BP
3:   for every v in G − R do
4:     clear CP
5:     CP.node = v
6:     FindBestParent(R, CP)
7:     FindBestChildren(R, CP)
8:     if CP.benefit > BP.benefit then
9:       BP = CP
10:    end if
11:  end for
12:  if BP.benefit > 0 then
13:    add BP.node to R and update R accordingly
14:  end if
15: until BP.benefit <= 0
```

We use plan variables to identify the view whose inclusion would most significantly lower the cost of tree scheduling. During each round, the loop beginning on Line 3 searches through the list of *candidate* views in $G - R$ and calculates the unique *inclusion cost* of each. We note that G represents a *guiding graph*, a data structure that captures the valid parent/child relationships for the views of a specific partial cube problem. Its construction will be discussed in Section 4.4.2. The inclusion cost is calculated by determining the most appropriate parent and children for the candidate. Inclusion will either result in a *net benefit* or a *net penalty* to the scheduling tree. If a benefit is identified, its value is compared to that of the current *best plan*; the current plan becomes the best plan if its benefit is greater than that of the current best plan. At the end of the current round the most cost effective view is added to R, using the relevant plan information. We then return to the loop on Line 1 to find the next view to add to R. This process continues until it is no longer possible to identify a view whose net cost is a benefit to the scheduling tree.

The key to the success of Algorithm 10 is the supporting methods FindBestParent and FindBestChildren. Before describing the two methods, we introduce the following notation. A node u can be created from v *iff* the attributes of u are a subset of the attributes of v. Let mode(v,u) be "scan" for $v, u \in G$ if u can be created from v via a scan, and "sort" otherwise. Let cost(v,u) be scan cost(v,u) if mode(v,u) = "scan", and sort cost(v,u) otherwise. Further, let RawDataSet denote the original data set and let parent(v,T) be the parent node of v in a given tree T. Finally, for a plan variable P, the procedure Clear(P) sets P.benefit to $-\infty$ and all other fields to NIL.

The first method, FindBestParent, is presented in Algorithm 11. The objective of the algorithm is to identify, for a given *candidate* node v, the least expensive node w already in R from which v can be computed. While the cheapest cost for the construction of v can be trivially guaranteed by ordering the attributes of v as a prefix of the smallest possible parent w — so that a simple scan of w can be used to compute

v — the computation of v in this manner would be inappropriate. Specifically, w may already have a scan child beneath it, one that has previously been identified as the most cost-effective scan child for w. If we blindly insert v as a scan child, we may break the current pipeline, significantly disrupting the tree beneath this point.

Algorithm 11 Find Best Parent

Input: Current tree R and a current plan CP.
Output: Sets the fields $CP.parent$, $CP.parentMode$ and $CP.benefit$ to represent best parent of $CP.node$.

{Initialize best parent to RawDataSet}
1: $CP.parent = RawDataSet$
2: $CP.benefit = 0 - cost(RawDataSet, CP.node)$
3: $CP.parentmode = mode(RawDataSet, CP.node)$
{Improve best parent, if possible}
4: **for all** w *in* $R - RawDataSet$ where the attributes of $CP.node$ are a subset of the attributes of w **do**
5: **if** w has no scan child AND $scancost(w, CP.node) < abs(CP.benefit)$ **then** {Case 1: $CP.node$ is inserted at the end of a pipeline}
6: $CP.parent = w$
7: $CP.benefit = 0 - scancost(w, CP.node)$
8: $CP.parentmode =$ "$scan$"
9: **else if** w has a scan child u AND $mode(w, CP.node) =$ "$scan$" AND $mode(CP.node, u) =$ "$scan$" AND $scancost(w, CP.node) < abs(CP.benefit)$ **then** {Case 2: $CP.node$ is inserted into an existing pipeline}
10: $CP.parent = w$
11: $CP.insertionscanchild = u$
12: $CP.benefit = 0 - scancost(w, CP.node)$
13: $CP.parentmode =$ "$scan$"
14: **else if** $sortcost(w, CP.node) < abs(CP.benefit)$ **then** {Case 3: CP.node become the start of a new pipeline}
15: $CP.parent = w$
16: $CP.benefit = 0 - sortcost(w, CP.node)$
17: $CP.parentmode =$ "$sort$"
18: **end if**
19: **end for**

To address this issue, we identify three distinct cases that the algorithm must consider, as illustrated in Figure 4.4. In the first case, we determine that the candidate

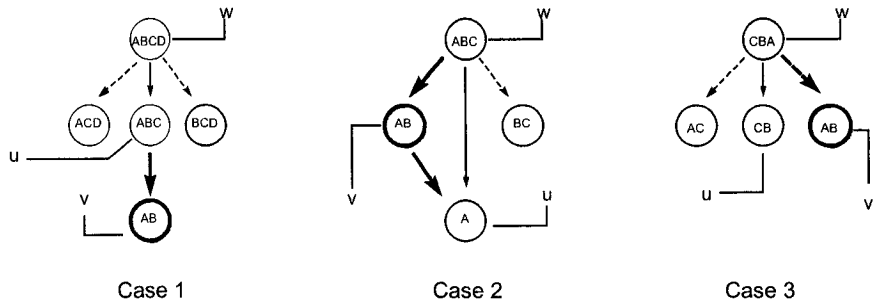


Figure 4.4: The three options for the insertion of AB in FindBestParent. Case (1) Pipeline tail insertion. Case (2) Pipeline scan insertion. Case (3) Re-Sort. Note: The emphasized lines represent new edges.

node v — here represented as the node AB — can be added as the *tail node* of an existing pipeline. This is in a sense the "best" solution since it is both cheap (no re-sorting is required) and simple (no nodes other than u, the current scan child, are affected). In the second case, we determine that the candidate can be inserted into an existing pipeline between two other pipeline nodes, a parent w and its current scan child u. This is also quite appealing from a cost perspective since additional sorting is not required. It is, however, somewhat more complex since a number of potential changes have to be recorded in the *current plan* object. In the final case, we accept the fact that the candidate cannot be incorporated into an existing scan pipeline, and we simply search for the smallest parent w to re-sort.

The second key method, FindBestChildren, is described in Algorithm 12. For a given node v, the method identifies the set of children that would create the largest benefit if they were created from the candidate node rather than their current parents in R. We begin by identifying the node in R that would best serve as a scan child for the candidate. Effectively, we have two choices for a scan child. First, if a "scan insertion" has been initiated in FindBestParent (i.e., case 2), then our scan child for the candidate node has already been identified in the *CP.insertionScanChild* field of the current plan object. If this isn't the case, then we look for potential child

nodes that are currently being sorted from other parents in R. We "steal" the one which, when converted into a scan child of the candidate, would provide the greatest savings. Finally, we examine the remaining potential children of the candidate, and select any that could be more efficiently sorted from the candidate than from their current parents. The best scan child and the list of new sort children are added to the current plan, and the benefit field is incremented by the savings associated with the new construction plan.

At the conclusion of a given costing round, the best plan is passed to a plan implementation method that physically adds the candidate to R. The ImplementPlan method takes its "instruction" from the plan data added by FindBestParent and FindBestChild. We consider R *fully reduced* once AddNonEssentialViews fails to produce a best plan with positive benefit.

4.4.2 Building the Complete Schedule Tree

In this section we describe a method for building the complete schedule tree, one that includes both essential and non-essential views. Note that in the previous section, we began with the assumption that the views of the selected set S had already been arranged within a tree of minimum cost, and that our goal was to add those non-essential nodes that would lower the cost of the original tree. Of course, in reality no such tree yet exists. Moreover, as noted by the authors of the PipeSort [105], an iterative bipartite matching algorithm cannot be used in the context of partial cube construction since parent/child pairings may be connected across arbitrary levels of the partial cube tree.

Recall that we used a greedy strategy to incrementally add non-essential views to the existing tree R. Our objective during each round was to identify those views which represented a net cost reduction, where only positive reductions — or benefits — were considered for inclusion in the tree. It is, however, possible to redefine the

Algorithm 12 Find Best Children

Input: Current version of the reduced tree R and a guiding graph G.
Output: Sets the fields $CP.scanChild$, $CP.sortChildren$ and $CP.benefit$ to represent best children of $CP.node$.

{Find the best scan child}
1: **if** $CP.insertionScanChild != nil$ **then**
2: $bestScanChild = CP.insertionScanChild$
3: **else**
4: **for all** w *in* R where the attributes of w are a subset of the attributes of $CP.node$ **do**
5: **if** $mode(parent(w, R), w) =$ "*sort*" **then**
6: **if** $cost(parent(w, R), w) - cost(CP.node, w) > bestScanChild\ benefit$ **then**
7: $bestScanChild = w$
8: $bestScanChild\ benefit = cost(parent(w, R), w) - cost(CP.node, w)$
9: **end if**
10: **end if**
11: **end for**
12: **end if**

{If no best scan child is found, we can return immediately since the candidate node will have no children in R}
13: **if** $bestScanChild == nil$ **then**
14: return
15: **end if**
16: $CP.benefit += cost(parent(w, R), w) - cost(CP.node, w)$
17: $CP.scanChild = bestScanChild$

{Find other children with positive benefit}
18: **for all** w *in* R where the attributes of w are a subset of the attributes of $CP.node$ AND $w \neq bestScanChild$ AND $mode(parent(w, R), w) =$ "*sort*" **do**
19: **if** $cost(parent(w, R), w) > cost(CP.node, w)$ **then**
20: $CP.benefit += cost(parent(w, R), w) - cost(CP.node, w)$
21: $CP.sortChildren += w$
22: **end if**
23: **end for**

concept of net benefit to include alternate costing strategies. Specifically, "greatest net benefit" may be redefined to include negative as well as positive benefits. In this context, the algorithm would effectively search for the view that would be associated with the "smallest penalty" during a given round. We note that in the special case of the AddNonEssentialViews method, the smallest penalty is exactly equivalent to the greatest net benefit.

Why would the redefinition of net benefit be useful? When constructing an essential spanning tree E it is likely the case that the cost of adding an essential view v exceeds the cost reductions it brings to potential children (if any children even exist). However, it is also true that while building E, it might be much more effective — during a given iteration — to incorporate v into an existing pipeline rather than an alternate essential view v'. In other words, even though both v and v' represent a negative benefit, the penalty associated with v is much smaller than that associated with v'. In this context, v is the best choice that we can make *at this point in time*.

Algorithm 13 describes the process of building the essential tree E. Logically, it is very similar to AddNonEssentialViews. The key difference is that instead of scanning the guiding graph G for a list of candidate nodes, the algorithm obtains them directly from S', a list that is initialized with the views in S. Moreover, during each iteration, the algorithm *must* add one of the nodes from S'. Because it may not be possible to find a node that actually reduces the current cost of E, the less restrictive costing model accepts the view that represents the smallest penalty. The algorithm terminates after exactly $|S|$ rounds, once S' is empty.

Now that we have described the process of adding both essential and non-essential views to the scheduling tree, we return to the problem of guiding graph generation.

Algorithm 13 Build Essential Tree

Input: Set S of selected group-bys.
Output: An essential schedule tree E consisting solely of the views in S.

```
    {Initialize E with nodes from S}
 1: S' = S
 2: E = emptyset
 3: while S' not empty do
 4:   clear BP
 5:   for every v in S' do
 6:     clear CP
 7:     CP.node = v
 8:     FindBestParent(E, CP)
 9:     FindBestChildren(E, CP)
10:     if CP.benefit > BP.benefit then
11:       BP = CP
12:     end if
13:   end for
14:   update E according to BP
15:   remove BP.node from S'
16: end while
```

Recall from Section 4.4.1 that a guiding graph G provides a description of the parent/child relationships which may be considered by the FindBestParent and FindBestChildren methods when attempting to insert non-essential views into the scheduling tree. In fact, we examine two distinct methods for producing G. In the first case, G is the spanning tree T produced by the original bipartite matching algorithm. As such we use the power of that technique to shape or guide the choices made by the greedy algorithm, the rationale being that bipartite matching has already identified cost effective relationships between various views in the lattice. The disadvantage, of course, is that with a partial cube many of the original views may not be selected. As such, it is possible that better pipelines exist within the partial cube context. Given this observation, we also examine a second guiding graph definition. Here, the guiding graph is simply the complete lattice L. In other words, the greedy algorithm is completely unrestricted in its choices — other than ensuring that child views contain a subset of the attributes of their parents.

It is now possible to present complete solutions for partial cube construction. Algorithm 14 provides the details of the technique that builds upon the lattice-based guiding graph. We begin by creating a guiding graph G from the lattice L. We then use BuildEssentialTree to construct from the views of S an essential spanning tree, and then add appropriate non-essential nodes with AddNonEssentialViews. The method EstablishAttributeOrderings — to be described shortly — identifies the scan pipelines implicit in R. Finally, the reduced scheduling tree is passed to the pipeline construction module that will actually generate the output. We note that this module is exactly the same as the one used to process the pipelines of the full cube.

Algorithm 14 Build Partial Cube: Lattice-Derived

Input: A lattice L and a set S of selected group-bys.
Output: A partial cube PC.

1: Prune L by deleting all nodes which have no descendent in S. Let G denote the result.
2: Execute BuildEssentialTree(E) to generate an essential spanning tree E.
3: Execute AddNonEssentialViews(G,E) to produce a reduced scheduling tree R.
4: Execute EstablishAttributeOrderings(R).
5: Build the partial data cube PC according to the reduced schedule tree R.

The reader will note the reliance upon two supporting algorithms in BuildPartial-Cube: Lattice-Derived. The first, in Step 1, is the method used to *prune* the Guiding Graph G created from the original lattice L. Before passing the PipeSort tree to the greedy algorithm, we want to ensure that it has been pruned of any unnecessary nodes. In short, we remove any node from the graph whose attributes are not a superset of at least one selected node. The process is described in Algorithm 15 and illustrated in Figure 4.5.

The second method, EstablishAttributeOrderings(R), is a post processing algorithm that has the task of identifying pipelines, or scan orderings, implicit in the reduced tree R. Note that, while all edges in R have been identified as either "scan" or "sort"

Algorithm 15 Graph Pruning Algorithm

Input: A lattice L and a set S of selected group-bys.
Output: A guiding graph G

```
1: for every node i in V - S do
2:   for all nodes j of S do
3:     if the attributes of i are a superset of the attributes of j then
4:       add node i to G
5:       break
6:     end if
7:   end for
8: end for
```

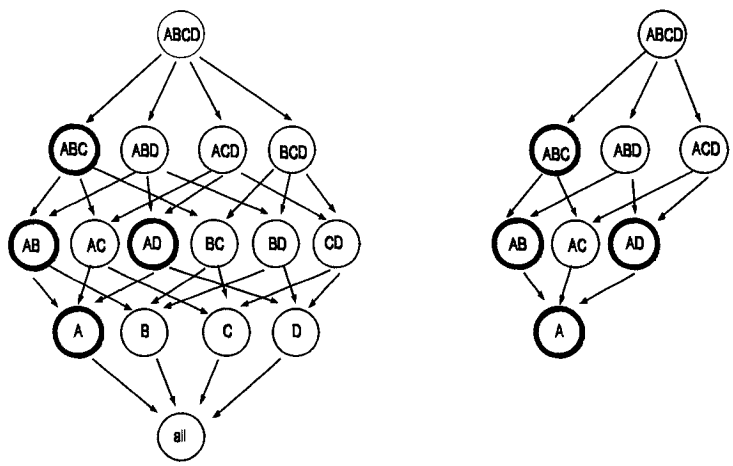


Figure 4.5: Graph Pruning, where bolded views belong to the selected set S.

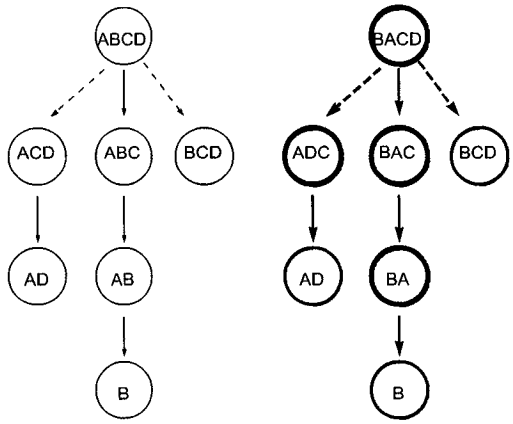


Figure 4.6: Illustration of EstablishAttributeOrderings(R). The emphasized nodes represent views whose attributes had to be re-arranged into a prefix order.

edges, the attribute orderings for the vertices, i.e. views, have yet to be established. The method EstablishAttributeOrderings(R) identifies all leaves in the reduced schedule tree R that are scan children. These leaves mark the bottoms of existing pipelines. For each such leaf u, a method FixAttributes(u) is called which recursively walks up the pipeline, starting at u. As the parent/child scan relationships are examined, the attribute order of the parent is modified to reflect the ordering of its child. For example, a pathway such as B — CB — CGB — DGBC would be re-ordered as B — BC — BCG — BCGD. See Figure 4.6 for an illustration.

As noted, the second alternative when constructing a guiding graph is to use the PipeSort spanning tree. Algorithm 16 describes this second partial cube approach. Besides the exploitation of the new guiding graph, the key difference is the post-processing method FixPipelines(R). This function has the task of (i) identifying nodes that have no scan child, (ii) creating a scan child for such nodes, and (iii) fixing the attribute orderings. Note that, since in Algorithm 16 the guiding graph is a subgraph of the PipeSort tree for the entire cube, the scan child u of a view v in the guiding graph may not be in R and therefore v may not have a scan child at this point. The method FixPipelines identifies all views v with at least one child but no scan child.

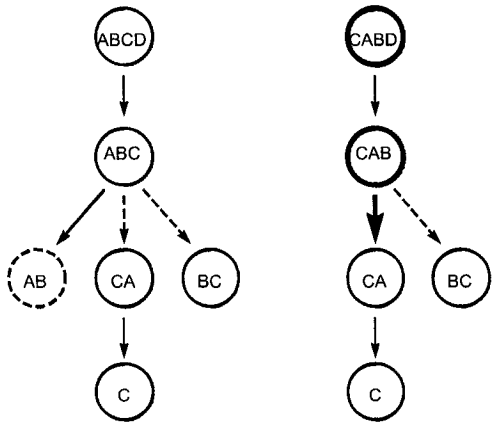


Figure 4.7: Illustration of FixPipelines(R). The dashed view is not included in the reduced tree. The bolded nodes represent views (i) whose attributes had to be re-arranged and/or (ii) were given a new scan child.

For each such node v, one arbitrary child u is made its scan child and FixAttributes(v) is invoked to correctly set the attribute orderings. See Figure 4.7. Note that it does not matter which child is chosen as the scan child since the scan cost is function of the size of v, not the child u.

Algorithm 16 Build Partial Cube: PipeSort-derived

Input: A PipeSort spanning tree T and a set S of selected group-bys.
Output: A partial cube PC.

1: Prune T by deleting all nodes which have no descendent in S. Let G denote the result.
2: Execute BuildEssentialTree(E) to generate an essential spanning tree E.
3: Execute AddNonEssentialViews(G,E) to produce a reduced scheduling tree R.
4: Execute FixPipelines(R).
5: Build the partial data cube PC according to the reduced schedule tree R.

This concludes the description of our basic partial cube method. To summarize, we provide a greedy method that makes locally optimal choices with respect to the inclusion of new spanning tree nodes. The algorithm is executed in two phases: (1) construction of an *essential* spanning tree and (2) transformation of the initial

spanning tree into a *reduced* spanning tree. We propose two alternatives for the implementation of the approach — an unconstrained lattice-based algorithm and a more "directed" PipeSort design. In both cases, the final reduced spanning tree is then processed by the existing pipeline architecture.

4.5 Analysis and Optimization

In Section 4.4, we described our algorithm for the construction of a partial cube scheduling tree. Specifically, we showed how to construct a tree in which we seek to minimize the costs associated with producing the views identified in the set S, plus any non-essential views that might lower the overall cost of the tree. In this section, we discuss the computational cost of producing this tree, and comment on the quality of results obtained by using the greedy method.

4.5.1 Complexity

We first provide an analysis of the running time of the new partial cube scheduling method (Algorithm 10 and Algorithm 13). While the processing of plan variables introduces a certain degree of "constant" overhead into the algorithm, we note that from an asymptotic perspective, the model can be reduced to a sequence of nested loops. Moreover, both component algorithms — BuildEssentialTree and AddNonEssentialViews — are structurally identical and thus will be bounded in exactly the same manner. In terms of the nested loops we can summarize their structure as:

(1) Repeat "best plan" costing iterations until no longer justified

  (2) Consider every candidate view during current iteration

    (3) For every candidate node, find best parent and best children

We note that each of these three "loops" has the potential to run in $O(n)$ time, where n refers to the number of views in the lattice. As such the upper bound on the partial cube algorithm is simply $O(n^3)$. How should we assess this bound? In many cases an $O(n^3)$ solution will be quite acceptable in practice. For example, in an eight dimensional space, the $O(n^3)$ algorithm requires at least $(2^8)^3 = 2^{24} = 16,777,216$ steps. While certainly not a trivial number of steps, the associated schedule tree can be computed in just a few seconds on a contemporary system of modest capability.

Still, in Section 4.3 we indicated that to be considered a success, any partial cube algorithm should be scalable to a larger number of dimensions. Consider, for example, $d = 14$. Our current $O(n^3)$ partial cube algorithm would run in $(2^{14})^3 = 2^{42} = 4,398,046,511,104$ steps. Clearly, this is too large a number of steps, even for more powerful computing systems. A more efficient solution is required.

4.5.2 Reducing the Cost of Building the Essential Tree

In this section, we will present a pair of alternative strategies for building the essential tree, each of which is capable of producing spanning trees in significantly less time than Algorithm 13. In Section 4.7, we will compare the quality of the results they obtain.

4.5.2.1 Recursive Pipeline Generation

We begin our search for a more cost effective solution by considering the simpler case of full data cube computation. Here, our objective is to construct a series of pipelines such that for $1 < i < d$, the various pipelines drawn from L allow for the most cost effective construction of the views at level i from those at level $i + 1$. Our general goal, of course, is to minimize the sorting costs associated with the construction of the views at level i. Because the cost of sorting is directly dependent upon the sizes of the "input" views at level $i + 1$, we want to identify a tree that uses the smallest views at each level when re-sorting is required. This, of course, is exactly the function

of bipartite matching.

In fact, there is an interesting general observation that can be made with respect to the construction of pipelines. Multiple views at level $i+1$ can share common children at level i. Likewise, each view v at level i can have a number of possible parents at level $i+1$. We note that the parents at level $i+1$ of a view v at level i must be at least as large as v. By extension, for a view v' at level i that is smaller than v, we can say that the *minimum required size* of its parents at level $i+1$ is strictly less than that of v. Why would this be important? The number of scan pipelines in a given lattice (full or partial) is finite — the minimal number being exactly equivalent to the number of views in the widest level of L, $\begin{pmatrix} d \\ \lceil d/2 \rceil \end{pmatrix}$. Since a minimum amount of sorting is required (each pipeline begins with a sort), and since re-sorting is the most expensive form of view creation, we would like to ensure that when sorting is performed, we are consistently using the smallest views for this purpose. Therefore, if parent views are connected via scan edges to views in the set Q of their largest children, we will eventually be left with "required sorts" for the views of M, the set of smallest children. Since, by definition, the *minimum required size* of the parents of the views of M is smaller than that of their larger siblings in Q, global sort costs should be significantly reduced.

This observation can be used as the basis of a new greedy algorithm for the construction of the *full* data cube. The details are provided in Algorithm 17. This is a *top-down* scheduling algorithm in the sense that we will be adding views to the essential tree E in a top to bottom order. Before explaining the new approach, it will be useful to introduce the notion of *free* views. By a free view, we mean a group-by that is found in the lattice L but that has not yet been added to the essential tree E. It is therefore a candidate for immediate inclusion.

The algorithm proceeds by first identifying the largest free view v in the lattice. Our objective is to build an entire pipeline beneath this point. We select the parent

w of v by identifying the *smallest* view w at the preceding level that contains a superset of the attributes of v. The view w becomes the "sort" parent of v. Next, we recursively descend down through the lattice, selecting a free view at each level. At each step we select the *largest* view amongst the available candidates and incorporate it into the current pipe. This process continues until the pipeline can be extended no further; i.e., there are no "free" children for the current *tail* node. We then execute the next iteration of the main loop, selecting the next largest view in the lattice and building another pipeline. The algorithm terminates once all the nodes of L have been added to E.

Algorithm 17 Optimized BuildEssentialTree

Input: A lattice L.
Output: An essential tree E.

```
1: Sort the views of the lattice by size, in descending order.
2: repeat
3:    Select the next largest "free" view v. This view will form the start point for
      the new pipe.
4:    smallestParent = NIL; smallestSize = maxINT
5:    for all "free" views w at previous level that contain a superset of the attributes
      of v do
6:       if sizeof(w) < smallestSize then
7:          smallestParent = w
8:          smallestSize = sizeof(w)
9:       end if
10:   end for
11:   Connect smallestParent to v with a "sort" edge.
12:   ExtendPipeline(v)
13: until all nodes have been added to E
```

In comparing the new optimized algorithm to bipartite matching, we note the following. By design, a greedy algorithm makes decisions based solely upon local benefit. While, in some cases, locally optimal choices are also globally optimal, for many problems this is not true. If the current local choices are *not* globally optimal, the choices available during subsequent rounds may be unacceptably poor. One of

Algorithm 18 Extend Pipeline

Input: A view v representing the tail node of an existing pipeline.
Output: An extended pipeline, with v as the new *tail* node.

```
1: largestChild = NIL; largestSize = 0
2: for all "free" views u at succeeding level that contain a subset of the attributes
   of v do
3:    if sizeof(u) < largestSize then
4:       largestChild = u
5:       largestSize = sizeof(largestChild)
6:    end if
7: end for
8: if no "free" views found then
9:    return
10: else
11:    Connect v to largestChild with a "scan" edge.
12:    Extend Pipeline(largestChild)
13: end if
```

the strengths of bipartite matching, in fact, is that it is able to consider costing "trade-offs." In other words, it may accept a slightly sub-optimal option for decision A if this leads to significantly better options for decisions B and C. In effect, the new algorithm attempts to incorporate a degree of this same logic into a greedy model. Specifically, it first seeks out the largest views as scan children so that when sorts are *subsequently* required, it is the smallest children — with parents of *minimum required size*— that will be used for this purpose. In this sense, we may say that the algorithm makes local decisions, but with a global influence.

We cannot of course guarantee that the recursive algorithm will always make decisions that are as effective as those of bipartite matching, since the latter is provably optimal between contiguous levels. For example, Figure 4.8 illustrates a case in which a greedy approach produces a sub-optimal number of scans (three versus four). In this case, the greedy algorithm initially selects three children for scans, none of which contain the attribute C. Because none of the remaining views contain this same attribute, the view C in the lower level must be computed with a sort. Bipartite matching, on the other hand, is able to identify a solution with four scan

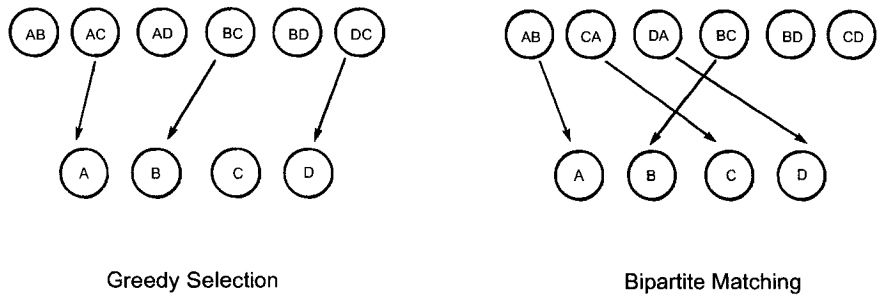


Figure 4.8: An example of how the greedy algorithm might identify fewer scan edges than the bipartite matching approach. In this case, the greedy method did not produce a scan edge for C.

edges. While this shortcoming of the new greedy algorithm might appear to be a significant problem, a closer analysis of this scenario suggests otherwise. Specifically, we note that when connecting level $i+1$ to level i, the parents views at level $i+1$ are the largest available (i.e., the algorithm always absorbs the largest children when building a pipeline). By extension, their potential children (i.e., the views at level i containing a subset of their attributes) are likely to be the largest children at level i. In the case of Figure 4.8, this would suggest that since the three largest parents all contain the attribute C, then the view C is likely to be quite large as well and would consequently have been taken first during pipeline construction. As a consequence, sub-optimal arrangements such as these, while possible, are unlikely in practice.

Having described the motivation behind the algorithm, we return to the original problem — complexity. It remains to be shown that the cost of the recursive greedy algorithm is in fact an improvement over the $O(n^3)$ run-time of the original approach. The new algorithm begins by sorting the n views of the lattice. We note that radix sort is not applicable here since the view weights are unrestricted in size. The $\theta(n \log n)$ sorting step is followed by a REPEAT loop, in which we add a new pipeline. Moreover, during each iteration of the main pipeline loop, we first select a "parent" view for the pipeline, then recursively extend the pipeline one view at a time. Now, for each of the $O(n)$ views included in this fashion, we must either find its best child and/or

its best parent. In the context of the full cube, there are $O(d)$ choices for each. As a result, we can conclude that the run-time of the new greedy algorithm is bounded as $O(n \log n + dn)$. We note that since $\log n = \log 2^d = d$, the bound can be re-written as $O((n * d) + dn)) = O(dn)$.

Of course, our objective in this chapter is to produce scheduling trees for partial cube problems. Recall that a bipartite matching model cannot be used for partial cube construction because of the fact that parent/child edges have the potential to skip levels in the lattice. Algorithm 17 and Algorithm 18, however, can easily be adapted to this environment. Specifically, we merely have to change Steps 5-10 of Algorithm 17 so that we consider *all* nodes above the current view v (with the appropriate attributes) as potential parents and Steps 2-7 of Algorithm 18 so that we consider *all* views below v (with the appropriate attributes) as potential children. The logic remains exactly the same. In short, we iteratively build up pipelines by incorporating the largest views into the pipes, this time allowing pipelines to skip levels. We note that *level skipping* can be utilized even on the full cube problem, something bipartite matching could not do.

In terms of cost complexity, we simply note that instead of $O(d)$ potential parents and children for each node v, we have $O(n)$ such candidates. Consequently, the run-time for the modified recursive partial cube algorithm is $O(n \log n + n^2) = O(n^2)$, still a significant improvement over the original $O(n^3)$ method.

4.5.2.2 An Aggressive Quadratic Time Algorithm

In this section, we describe an alternate approach to building an essential tree that relies upon an aggressive search for the best parent of a new view. Recall that in the preceding section, pipelines were extended by recursively incorporating the largest available child view into the pipeline. The model assumed that once a view was added to the pipe, we would *always* want to use it as a scan parent in the succeeding iteration.

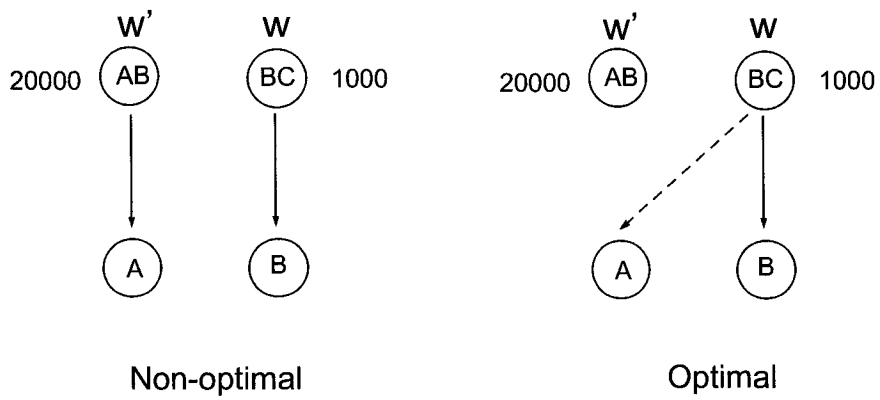


Figure 4.9: An illustration demonstrating that the minimal number of sorts may not always be optimal.

In fact, this may not always be the case. Figure 4.9 demonstrates that when there are large imbalances in size between parents, it may occasionally be cheaper to sort a parent w rather than scanning a larger "free" view w'. Again, bipartite matching is able to identify this sort of situation, while the recursive pipeline algorithm is not.

With respect to practical applications, we make the following general observation.

Observation 1. *For a lattice L and a selected set S, the impact of counter-productive scans decreases as the ratio of $|L|/|S|$ decreases.*

This is the case because, with a complete cube, the weight of the extracted spanning tree is disproportionately represented in the top of the graph where the large sparse views are located. Furthermore, differences in sizes between potential parents are small since extreme view sparsity produces views that are all very close in size to that of the raw data set. As a result, it is extremely unlikely that sorts will be less expensive than scans in this part of the tree. Though this situation changes as we move towards the base of the tree — as depicted in Figure 4.9 — the construction cost of these views represents a relatively small portion of the global cost.

Partial cubes, on the other hand, can present a different scenario. Since the goal is often to eliminate many potential parents views, it is possible that there may be

wider differences between the sizes of potential parents since these parents may come from many different levels. In such cases, it may be cheaper to sort a parent w than to scan a much larger parent w'. For this reason, we present a second greedy algorithm, Algorithm 19, that is designed to specifically address this concern. Here, we first sort the views in descending order by size. Working from largest to smallest, we add each view v to the spanning tree. When adding, we only concern ourselves with how to connect v to its most appropriate parent w. The parent is chosen by "aggressively" searching for the very best parent at this point, where the best parent is equated with the cheapest possible scan or, if no scan is possible, with the cheapest possible sort. Note the difference between this algorithm and the recursive pipeline version. In the latter case, large parents "absorbed" large children, leaving smaller parents available for smaller children. In the aggressive version, very large scans will be avoided in favour of smaller sorts. The risk of, of course, is that we will "use up" the available scans prematurely — by missing the opportunity to use slightly larger parents for scan pipelines — leaving us to sort unnecessarily. The experimental results presented in Section 4.7 will compare the two techniques on practical problems.

4.5.3 Reducing the Cost of Adding Non Essential Views

In this section, we describe a more efficient approach for adding non essential views to the initial spanning tree. We note that while the algorithm for building the essential tree, described in the previous section, runs in $O(n^2)$, the run-time of the "combined algorithm" is still $O(n^3)$ since the current version of AddNonEssentialViews requires $O(n^3)$ time. In other words, we must be able to produce an $O(n^2)$ version of AddNonEssentialViews; otherwise, a quadratic time BuildEssentialTree algorithm provides no advantage in an asymptotic sense.

We propose a technique by which views can be added directly to the tree, without the need to review all candidate nodes before selecting a single "best view" in

Algorithm 19 Aggressive Essential Build

Input: A lattice L and a selected set S.
Output: An Essential scheduling tree E.

```
1: Sort the views of S in descending order
2: repeat
3:    Get largest "free" view v in S
      {Initialize smallestParent variables}
4:    smallestSortParent = smallestScanParent = NIL
5:    smallestScanCost = smallestSortCost = intMAX
      {aggressively add next largest "free" view}
6:    for all possible parents w of v do
7:       if w has no scan child AND scanCost(w) < smallestScanSize then
8:          smallestScanParent = w
9:          smallestScanCost = scanCost(w)
10:      end if
11:      if sortCost(w) < smallestSortCost then
12:         smallestSortParent = w
13:         smallestSortCost = sortCost(w)
14:      end if
15:   end for
16:   add v to E
17:   if scan child was found then
18:      connect smallestScanParent to v with a "scan" edge
19:   else
20:      connect smallestSortParent to v with a sort edge
21:   end if
22: until all views in S have been added to E
```

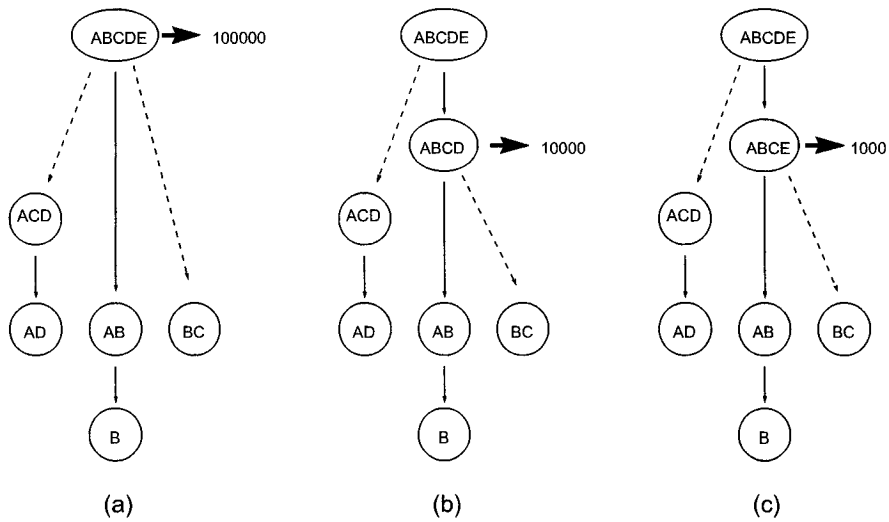


Figure 4.10: Significance of the order of addition. Case (a) shows the original tree. In case (b) we decide to add ABCD with a scan insertion. However, case (c) demonstrates that a more cost effective solution was actually available.

a round-by-round manner. In so doing, we collapse the outer loop of the original AddNonEssentialViews algorithm in order to convert an $O(n^3)$ algorithm into an $O(n^2)$ one. Specifically, the new algorithm greedily adds candidate views as soon as it determines that a new node has the potential to produce *any* reduction in the global tree cost. We will not wait to find the very best new node, but instead will add views with a net benefit immediately. The obvious question, of course, is "does the order of addition matter?" The simple answer is "yes." Clearly, it is possible to commit to an alteration to a pipeline — say with a scan insertion — only to later find a much smaller non-essential view that could have more cheaply supported the same child set. Figure 4.10 provides a simple example.

Consider the following three policies with respect to ordering the inclusion of beneficial free views:

1. Consider views in a first-come, first-serve order.

2. Identify the largest viable free view (top-down).

3. Identify the smallest viable free (bottom-up).

It should be clear that the first choice, first-come first-serve, is effectively equivalent to the approach depicted in Figure 4.10. As such, it cannot be expected to produce consistently good results.

The remaining two-policies consider views in an order based upon size. We note that this is not the same as *adding* views based upon size — as was done with the optimized BuildEssentialTree algorithm. Here, our goal is to re-structure the search strategy in such a way that the most beneficial views are likely to be examined first. As one option, we may choose to consider the largest views first. Because views decrease in size towards the bottom of the tree, this approach can be described as a top down search strategy. Conversely, the smallest-to-largest model represents a bottom-up traversal of G.

While it might not be immediately obvious how the two approaches would differ, we observe the following.

Observation 2. *The potential for error when greedily adding non-essential views is most pronounced when selecting views from the guiding graph in a top down fashion.*

There are four facts that support this observation.

1. Adding the largest nodes first can clearly result in the non-optimal situation depicted in Figure 4.10.

2. The views in the upper lattice are simply far larger than the views lower in the graph; unnecessary additions therefore represent a larger penalty relative to the global cost.

3. Costing decisions are more significantly impacted by the accuracy of the "best children" phase than the "best parent" phase in that a candidate may have many children but only one parent.

4. If views are sorted and then evaluated by size, we can be certain that all possible children of v have been added to R by the time v is considered for inclusion since a view u can only be a child of v if $size(u) \leq size(v)$.

If we proceed top-down, then, we must make greedy decisions about the additions of a node v without an appreciation of possibly cheaper alternatives. Moreover, we must do so without knowing the final state of the tree below v. Conversely, if we move bottom-up, we are presented with the cheapest alternatives first and, perhaps more importantly, the decisions regarding "best children" will be more accurate since we are guaranteed to have complete knowledge of the views beneath v.

Given these observations, we now present a new algorithm for adding non essential views. Details are provided in Algorithm 20. As was the case with the original AddNonEssentialViews algorithm, we take as input an essential spanning tree E and a guiding graph G. To begin, the views of G are sorted by size, this time in *ascending* order. Once the sorted list is created, we proceed by processing each candidate v in terms of the existing FindBestParent and FindBestChildren methods. If a positive benefit is calculated, we add v to R immediately and move on to the next view in the list. The algorithm terminates when we have examined all views from G.

Algorithm 20 Optimized Add Non Essential Views

Input: An essential spanning tree E and a guiding graph G.
Output: A reduced scheduling tree R.

```
1: Sort the views of G in ascending order by size.
2: repeat
3:    clear CP
4:    CP.node = v
5:    FindBestParent(R, CP)
6:    FindBestChildren(R, CP)
7:    if CP.benefit > 0 then
8:       add CP.node to T as per CP plan description
9:    end if
10: until all views from G have been considered
```

With respect to the cost complexity of the new model, note that the algorithm consists of a sorting phase, followed by a view inclusion phase. The sorting phase requires time $\theta(n \log n)$, where n is equivalent to the number of views in G. In the inclusion phase, we simply execute the $O(n)$ BestParent/BestChild combination for each of the $O(n)$ views in G. The result is an $O(n^2)$ bound for the second phase. With this cost dominating the $\theta(n \log n)$ cost of the sort phase, we conclude that the run-time of the new algorithm has an upper bound of $O(n^2)$. We can therefore say that we have accomplished our primary performance objective — *both* BuildEssentialTree and AddNonEssentialViews run in $O(n^2)$, giving us a quadratic time solution for partial cube construction. Proposition 4 quantifies the run-time advantage of the new algorithms.

Proposition 4. *For a partial cube problem whose input is an n-node graph, the run-time of an $O(n^2)$ solution will exceed that of an $O(n^3)$ solution if the dimension count for the quadratic solution is increased by 50%.*

Proof. We need to determine the degree to which d must be increased so that the cost of a quadratic time algorithm using this larger value for d will be $\geq$ the cost of a cubic time algorithm on the original value of d. Let us denote this *multiplicative factor* as β. Further, we note that in the context of the data cube, the input size n for a d-dimensional data cube can be equivalently expressed as 2^d. We are therefore interested in solving the following inequality:

$$(2^{d\beta})^2 \geq (2^d)^3$$

$$2^{2d\beta} \geq 2^{3d}$$

$$\log(2^{2d\beta}) \geq \log(2^{3d})$$

$$2d\beta \geq 3d$$

$$\beta \geq 3/2$$

□

The practical significance of Proposition 4 is that it gives us a precise means by which to determine the direct benefit that our new solution provides on real

architectures. For example, experimental testing has shown that the $O(n^3)$ solution is *practically feasible* on the Linux cluster at between eight and ten dimensions. This would imply then that the new $O(n^2)$ algorithm would be feasible in the range of 12 to 15 dimensions on this same machine.

Finally, we note that in addition to confirming a performance advantage, we must also assess the quality of the trees generated by the new methods. This evaluation will be provided in Section 4.7.

4.5.4 Extending the Algorithm into High Dimensions

An $O(n^2)$ algorithm for the partial cube allows us to handle higher dimension problems than would otherwise have been possible. In this section we explore the problem of extending these algorithms to work in yet higher dimensional spaces.

Given the structural approach that we have taken with our partial cube methods — for each of $O(n)$ views, the algorithms find best parents and best children from $O(n)$ candidates — it is unlikely that a sub-quadratic time version of our current algorithms is possible. As such, we must take another approach to run-time reduction in high dimensions.

If we cannot reduce the asymptotic complexity of the algorithm, then the obvious alternative is to reduce the input size n. With most computational problems, this is of course not possible since the input size is an unalterable parameter of the problem instance. In the context of the partial cube, however, we have already described a pruning algorithm that reduces the size of the *candidate space* prior to execution of the main loops. Our goal, then, is to improve upon the pruning technique to further reduce the size of the candidate set without significantly compromising the quality of the solution.

Recall that the objective of the original $O(n^2)$ pruning algorithm was to remove any candidate nodes that could not possibly serve as an ancestor of any of the selected

views in S. We note that as the size of S increases, the ability of the algorithm to prune candidates diminishes significantly since it becomes much more difficult to find views that cannot be an ancestor of at least one view in S. In fact, for one of the most important partial cube special cases, the current pruning algorithm is *guaranteed* to prune nothing. Specifically, if we construct S such that it contains only those views *below* a certain level in the lattice, it is not possible to find any candidates that are not ancestors of at least one view in S. If this is not clear, consider that the first (i.e., lowest) level of the lattice contains d views, each consisting of one of the d attributes in the problem space. Clearly, *all* candidates must be made up of combinations of these same attributes and can thus theoretically serves as ancestor views for one or more views in S.

Our approach therefore is to develop a different mechanism for dealing with high dimensional spaces. To begin, we re-iterate that a candidate node is added to the current spanning tree *if and only if* the cost of its physical creation is less than that of the savings it bring to the generation of its potential children.

Theorem 5. *A candidate node v will not be added to the spanning tree R if it cannot improve the construction cost of at least two child nodes already in R.*

Proof. Let us assume that a child node u in R currently has a parent node w. Further, let us assume that a candidate v is inserted into R and replaces w as the parent of u, and that u is the only child of w. There are in fact two cases in which v can become a parent. First, it can be used as a scan parent. In other words, it can be inserted into an existing pipeline with a scan insertion. Recall from Chapter 3 that the scan cost of u is bounded by the size of its parent, in this case $|w|$. Though the new cost of computing u from v may be smaller, since $|u| \leq |v| \leq |w|$, the scan cost for v is once again bounded by $|w|$. Consequently, scan insertion can never be beneficial if v has just one child.

In the second case, v replaces w as a sort parent of u. If v is smaller than w, then u can clearly be sorted more cheaply from v. However, we must now include the cost of computing v. If v is not inserted via a scan insertion (described above), it *must* be constructed with a sort of its new parent q. In other words, it cannot be inserted at the end of an existing pipe. If that were true, then u could have previously been inserted at the end of this same pipe, again making this a scan insertion case. As such, v must be sorted from a parent q that is at least as large as w. If this were not the case, q would have been the sort parent of u, not w. The cost of adding v with

a sort is therefore the cost of computing u as a scan of v, plus the cost of re-sorting q to produce v. Since q is at least as large as the original parent w of u, the cost of computing v is at least as large as the original cost of generating u. Again, we can conclude that this form of view inclusion cannot lead to a net benefit.

□

Theorem 5 demonstrates that in order for a non-essential view to be added to R, it must become the parent of at least two children. Since a given parent can only have one scan child, the remaining children would be computed by re-sorting v.

We note as well that the likelihood of inclusion in the spanning tree is a function of the position of v within the guiding graph. Observation 3 formalizes this notion.

Observation 3. *As we move from the bottom to top of G it becomes progressively more unlikely that a candidate v will be able to reduce the global tree cost.*

The veracity of the observation rests upon an understanding of the sparsity of candidate views. As we move upwards through G, we incrementally increase the number of dimensions in v. Recall that as dimension count increases, so too does the sparsity of the view. Eventually, v will be almost identical in size to its potential parent views. This is significant since parent views consist of a superset of the attributes in v and can thus also serve as parents for any of the potential children of v. In other words, the value of v as a non-essential addition declines as we move towards the top of G since it is increasingly unlikely that it will be more cost effective than any number of other candidates. Figure 4.11 depicts alternative scenarios for the inclusion of a particular view, ABC. In case (b), we see a scan insertion in which the candidate for inclusion, ABC, is considerably smaller than its parent, $ABCD$. This example is typical of the situation in the lower, more dense, portion of the lattice. Clearly, there is an advantage to adding ABC in this case. Conversely, in case (c), we have an ABC that is very close in size to its parent $ABCD$, a situation that is very common in the upper regions of the lattice. Here, it is actually a penalty to add ABC.

Algorithm 21 incorporates Theorem 5 and Observation 3 into an effective heuristic solution. Beginning at the base cuboid, the method moves downwards through the

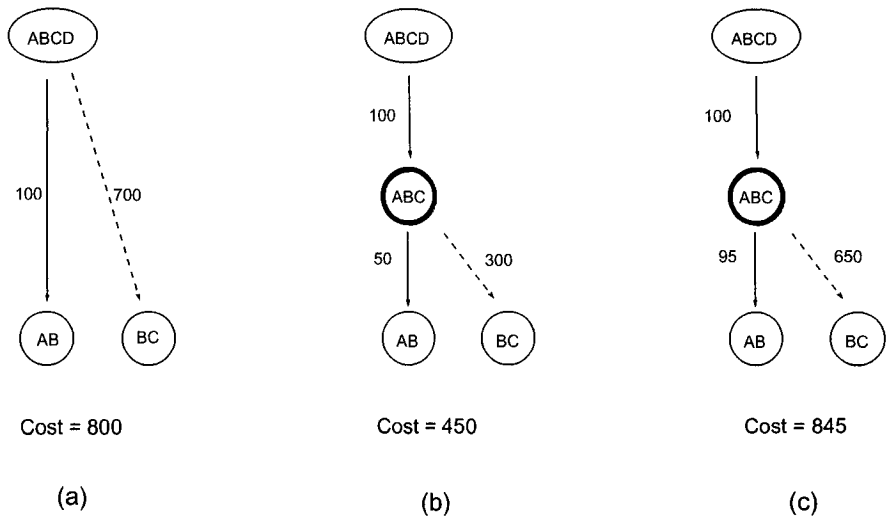


Figure 4.11: Pruning ineffective nodes. Case (a) shows the original tree. In case (b), a more dense ABC node offers great benefit. However, in case (c), we see that a sparse ABC node actually increase the cost.

lattice, selecting potentially "useful" views and adding them to the guiding graph G that will be used to provide candidate nodes to our greedy algorithm. The logic for inclusion is as follows. For a given candidate node v in L, we will assume that its largest parent is already in G. We do this since (i) we know that any parent of v can also serve as a parent of the children of v, and (ii) the largest parent would produce the maximum possible benefit for v. Now, we have already determined that the inclusion of v requires that v have at least two child nodes. Furthermore, one of these will be created with a scan and the other with a sort. Though there are a number of options for the creation of v (i.e., scanning and sorting), we again want to ensure that we assess the potential of v in terms of its maximum possible benefit. We therefore assume that v can be added simply with a scan insertion (as depicted in Figure 4.11). Now, let us also make the assumption that v will be used as a *replacement parent* for an existing parent w, and further that this will only happen if w currently has two current children. Consequently, we can say that the current cost of computing these two children is $scanCost(w) + sortCost(w)$, while the cost associated with the

replacement parent v is $scanCost(w) + scanCost(v) + sortCost(v)$. Since a scan of w is common to both functions, we can conclude that in this scenario, v will only be useful if $scanCost(v) + sortCost(v) < sortCost(w)$.

Algorithm 21 Pruning for High Dimensions

Input: A lattice L, and a *confidence factor* β.
Output: A guiding graph G.

```
1: for all candidate nodes v in L do
2:    From the O(d) potential parents in L, find the smallest parent w of v
3:    if scanCost(v) + (β * sortCost(v)) ≤ β * sortCost(w) then
4:       add v to G
5:    end if
6: end for
```

This is exactly the logic utilized in Algorithm 21 to determine if a view v should be added to the guiding graph. For the sake of flexibility, the metric is extended by way of a user-defined *confidence factor* β that determines how aggressively the algorithm will prune the lattice. A β value of one implies the assumption that v will only have two children. With increasing values of β, the algorithm becomes more conservative in that it allows for the possibility that v may have many children. In this regard, we note that even a minor size difference between the original parent w and the candidate v eventually produces enough savings in the sorts to outweigh the added scan cost of v. The confidence factor therefore allows the user to determine his/her degree of risk, that is, the willingness to occasionally prune a candidate that may actually have some benefit.

We re-iterate that this new pruning technique is a heuristic solution. In other words, it should be understood that we are trying to efficiently *approximate* the guiding graph G with a small number of views. As noted, it is possible that a small β factor might occasionally result in the pruning of a candidate that could actually provide some benefit. Proposition 5 defines the potential error.

Proposition 5. *For a view v that has been unnecessarily pruned by Algorithm 21, the maximum error is bounded as $(k-1)scanCost(v)$, where k is the number of sort children that v would have had if included in the guiding graph.*

Proof. If we assume the most aggressive confidence factor (i.e., $\beta = 1$), then we know that v was pruned because $scanCost(v) > sortCost(w) - sortCost(v)$. This could only have produced an error if the number of sort children k for v was actually greater than one. So while $scanCost(v) > sortCost(w) - sortCost(v)$ is correct, $scanCost(v) > k(sortCost(w) - sortCost(v))$ for $k > 1$ is not. Thus the error is bounded as $(k-1)scanCost(v)$.

□

We observe that while the potential error is real, the risk of producing it in high dimensional spaces is low when the size of the selected set is small relative to 2^d. This is the case simply because the restricted size of the selected set S in partial cube problems rarely results in parents with large child counts. Consequently, small confidence factors are unlikely to unnecessarily prune many "useful" views. This observation is supported by the experimental results in Section 4.7.

Finally, we address the issue of the cost complexity for the pruning algorithm. Recall that the original pruning algorithm was an $O(n^2)$ solution. In fact, even if that algorithm were more effective it would be worthless in the current context since the objective of the new algorithm is to reduce the input space so that an $O(n^2)$ greedy partial cube algorithm can tackle larger problems. If the pruning algorithm itself is $O(n^2)$ on the n nodes of the full lattice, then we accomplish nothing by pruning.

An examination of the logic of the new algorithm demonstrates that we make a linear pass through the lattice, looking for views to prune. At each step, we check the $O(d)$ possible parents of v. Note that because we are pruning the complete lattice, there are only $O(d)$ possibilities, not $O(n)$. Thus the upper bound is $O(d * n)$.

It is perhaps useful to conclude this section by placing the optimized algorithms into a single context. We began by describing new algorithms, Algorithm 17 and Algorithm 19, for the generation of the initial spanning tree. While these algorithms runs in $O(n^2)$, it is important to understand that the input n in this case represents

only the views in the selected set S, not the full lattice. Where $|S|$ is small, it is likely to be fast in practice. The second algorithm, Algorithm 20, is also $O(n^2)$, but takes as input the views in the set $L - S$. Since this space is potentially very large, we use the new pruning algorithm, Algorithm 21, to reduce the input size. Note that the only component algorithm of the partial cube method that actually computes over the complete lattice is the pruning algorithm, and though it actually processes all of the lattice, it does so in $O(dn)$ time.

4.6 Parallel Partial Data Cubes

Thus far, we have described the process of generating a partial cube scheduling tree for uni-processor implementations. As noted previously, the pipelines of this tree can then be processed in exactly the same manner as those of a full cube. Recall from Chapter B that the parallelization of the full data cube is based upon a partitioning of the underlying scheduling tree. Since the partial cube algorithm also produces a scheduling tree, it is a relatively simple task to produce a parallel version of the algorithm, one that builds completely upon the existing data cube infrastructure. Algorithm 22, ParallelPartialCube(p, S, PC) describes the extended model. In short, our approach is to (1) generate the reduced partial cube schedule tree R, (2) partition R into subtrees representing workloads of equal size and (3) distribute the workload over the p processors $P_1, \ldots, P_p$. We note that this model is applicable to any of the partial cube scheduling algorithms discussed in this chapter.

We note that by exploiting a "generic" parallel model, we avoid having to design two distinct parallel data cube algorithms, one for full cubes and the other for partial cubes. In effect, we have restricted the necessary changes to the sequential component of both models. Given that parallel algorithm design can be more complex and time consuming than similar work in a sequential context, the ability to re-use this generic architecture is a significant benefit of our research.

Algorithm 22 Parallel Partial Cube

Input: Set S of selected group-bys, a number of processors, p, an over-sampling factor s, and a guiding graph G.
Output: Distributed partial data cube PC_{dist}.

```
1: for processor P_0: do
2:    Prune G by deleting all nodes which have no descendent in S. Let G denote
      the result.
3:    Execute BuildEssentialTree(E) to generate an essential spanning tree E.
4:    Execute AddNonEssentialViews(G,E) to produce a reduced scheduling tree R.
5:    Execute EstablishAttributeOrderings(R).
6:    Execute TreePartition(R, p, s) to produce p sub-tree subsets Σ_1, ..., Σ_p.
7: end for
8: for all processor P_i, in parallel: do
9:    Compute all group-bys in subset Σ_i on processor P_i according to the relevant
      schedules.
10: end for
```

4.7 Experimental Evaluation

In this section, we present experimental results for the partial cube algorithms discussed throughout the chapter. We will focus on both run-time performance of the algorithms and the quality of the schedule trees produced. Unless otherwise stated, the default parameters and testing methodology for the experiments described in this chapter are the same as Chapter 3.

We will present two distinct types of results. In the first case, we look directly at the quality and run-time performance of the new partial cube algorithms. In particular, we will compare the "optimized" scheduling tree algorithms with both the non-optimized cubic time versions and the original bipartite matching algorithm. In the second case, we look at the performance of the complete parallel partial data cube framework, including not just schedule tree generation but partitioning and view construction as well.

4.7.1 Evaluation of Schedule Tree Generation Algorithms

In this first section, we examine the performance of the sequential schedule tree component. Though we could evaluate these algorithms on any single-processor machine, for consistency we utilize a single CPU on the local Linux cluster, described in Chapter 3, for this purpose.

4.7.1.1 Quality of Generated Trees

Recall that while it is not possible to identify an optimal solution for partial cube schedule tree construction (at least not in polynomial time) we do have a full cube algorithm — based upon bipartite matching — that is known to produce extremely good spanning trees for all 2^d views [105]. As such, by running the partial cube algorithms on the complete lattice, it is possible to experimentally study the quality of their solutions relative to the best known comparable algorithm from the data cube literature.

Figure 4.12 describes the quality/weight of the spanning trees generated by the new partial cube algorithms relative to the trees produced by bipartite matching. Note that *all* of the greedy partial cube algorithms produce trees that are less than six percent larger than those of bipartite matching for dimension counts between six and twelve. This is a significant observation in that it clearly justifies the choice to employ a greedy model in the current context.

With respect to the partial cube alternatives, the results suggest that the best trees were consistently produced by the algorithm based upon recursive pipeline construction. Recall that this is a quadratic time algorithm, demonstrating that the fastest algorithm *also* produces the best trees. In fact, from six to 12 dimensions, the trees produced by this $O(n^2)$ technique were less than one tenth of one percent larger than those generated via bipartite matching. As such, we can conclude that

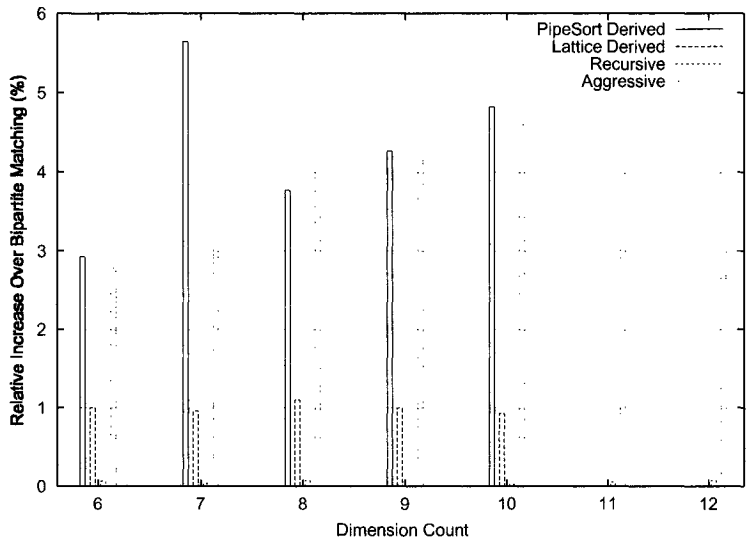


Figure 4.12: The cost of the spanning tree in relation to the cost of those generated by bipartite matching. The cubic time algorithms were not computed for 11 and 12 dimensions.

the "globally influenced" local decisions of the recursive algorithm appear to be somewhat more powerful than the purely local decisions of the original cubic time greedy models.

Finally, we note that results were not charted for the cubic time algorithms beyond 10 dimensions. This is simply because the run time in these larger spaces made testing too time-consuming. This is an issue addressed in the following section.

4.7.1.2 Run Time Performance on the Full Cube

Figure 4.13 depicts the run time for the four partial cube alternatives and bipartite matching on the test cases depicted in the previous section. The reader should note the use of a logarithmic axis to represent run time. This was necessary since the run-time for the $O(n^3)$ algorithms grew extremely quickly beyond eight dimensions. For example, at 10 dimensions the run-time for the cubic time algorithms was just over 2800 seconds, with the non-cubic time algorithms completing in under 20 seconds. Beyond 10 dimensions, we used extrapolation to estimate the run time for the cubic

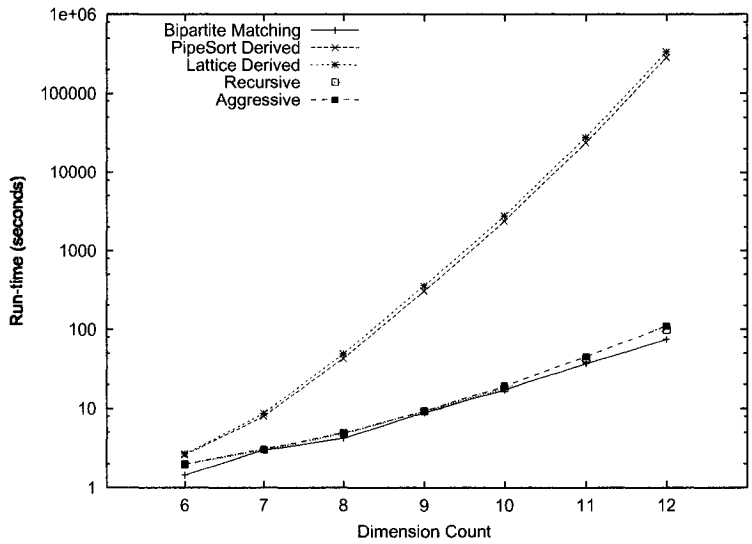


Figure 4.13: Run time performance for schedule tree generation on the full cube. At 11 and 12 dimensions, the times for the cubic time algorithms are estimated.

time algorithms. At 12 dimensions, they are expected to take more than 330,000 seconds (approximately 3 days, 19 hours). In this same space, the three non-cubic time algorithms produced their spanning trees in just over one minute. The differences at higher dimensions, of course, become even more striking. For example, at 14 dimensions the comparison would be 14 months ($O(n^3)$) versus 5 minutes ($O(n^2)$). Finally, it is interesting to note that the performance curves for the quadratic algorithms are very similar to that of the original bipartite matching algorithm.

4.7.1.3 Computing Partial Cubes

In this section, we compare the schedule trees produced by the four partial cube algorithms on selected sets of view. The quality of the final tree is represented relative to a baseline solution. In the current context, the baseline represents the cost/weight of a tree in the absence of a partial cube algorithm. In fact, without such an algorithm, there are essentially two possible approaches to build a partial cube: (1) build the full data cube and then return the selected views only, or (2) calculate each of the selected views by a separate sort of the raw data set. Which of these two approaches

is better depends essentially on the percentage of selected views. For a small number of selected views, the individual sorts will often be faster, while building the full data cube is often a better option when the percentage of selected views is high. Algorithm 23, which always selects the faster of these two approaches, will be used as the baseline against which the four partial cube algorithms will be compared.

Algorithm 23 Simple Partial Cube

Input: Set S of selected group-bys.
Output: Partial data PC.
1: Compute the Pipesort spanning tree of full lattice L.
2: Compute the cost of an individual sort and scan of the raw data set for each view in S.
3: Select the cheapest alternative from Step 1 and Step 2.

To evaluate the partial cube algorithms, we have randomly selected subsets consisting of 10%, 25%, 50%, and 75% of the views in the full space (Note that the selected percentage cannot be equated directly with the work to be performed. For example, given the widely varying weights of each view, a randomly chosen subset of 25% of the views could represent 50% of the total weight of the lattice). Because we are interested in comparing the quality of the quadratic time solutions to the cubic time solutions, we restrict ourselves to dimension counts in the range of six to nine. Figure 4.14 presents the results. For the 10%, 25%, and 50% selections, we note the gradual decrease of relative weight reduction as we move toward higher dimensions. This is to be expected since an increase in dimension count implies an increase in the sparsity of intermediate views. Since an increase in sparsity in turn implies an increase in size relative to the raw data set, we can conclude that the cost savings will decline slightly in higher dimensions since the cost reductions related to the sorting/scanning of intermediate pipeline views are less dramatic. Nevertheless, cost savings of between 20% and 60% are generated by the new algorithms.

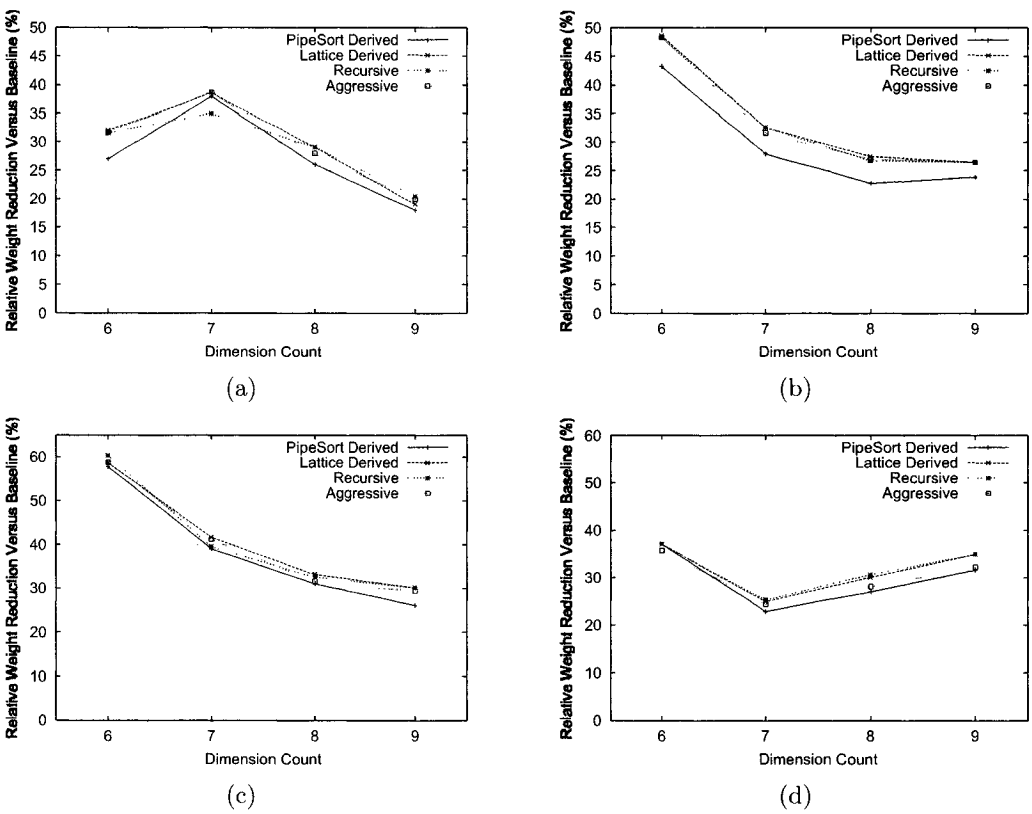


Figure 4.14: Relative weight reduction for the schedule trees produced on subsets of size (a) 10% (b) 25% (c) 50% (d) 75%. The baseline in this case is chosen as the smaller of (i) a sort of the raw data set for each view or (ii) computation of the full cube.

At 75%, the situation is slightly different. Here, the partial cube problem starts to look more like a full cube problem. Consequently, given the large number of views to compute, the relative cost reduction starts to rise as we move into larger spaces. Savings of between 25% and 35% are depicted.

Beyond the baseline comparisons, it is also important to note that there is relatively little difference between the cost reductions generated by each of the greedy algorithms. Moreover, across the four graphs, we can discern a slight advantage for the quadratic time recursive algorithm, a result in keeping with the trend demonstrated for the full cube evaluation. We do note that the "aggressive" quadratic algorithm does provide some benefit for lower dimension counts on small subsets. In fact, this was exactly its objective. Specifically, when many views are absent from the guiding graph, it is expected that automatically choosing the largest children for inclusion in scan pipelines might occasionally lead to unnecessarily large scans. The aggressive algorithm avoids these "mistakes" by selecting the smallest computable child at every point. Thus, it is able to produce slightly better trees on small dense subsets. Nevertheless, as a general purpose algorithm — one combining performance and consistency — the quadratic time recursive algorithm would appear to be the clear choice.

Finally, we note that the use of randomly selected subsets may underestimate the cost savings for an important class of partial cube problems. Specifically, in high dimension spaces, users and administrators often want to select the majority of views from the lower portion of the lattice since such views are more intuitive to visualize and interpret. Figure 4.15 illustrates the relative cost reductions when the selected views are limited to those containing three attributes or less (the algorithm, of course, is free to add larger, non-essential views). For simplicity, we restrict the experiment to the quadratic time recursive algorithm. Under these circumstances, the algorithm consistently reduces the weight of the schedule tree by 60% to 70% for dimension

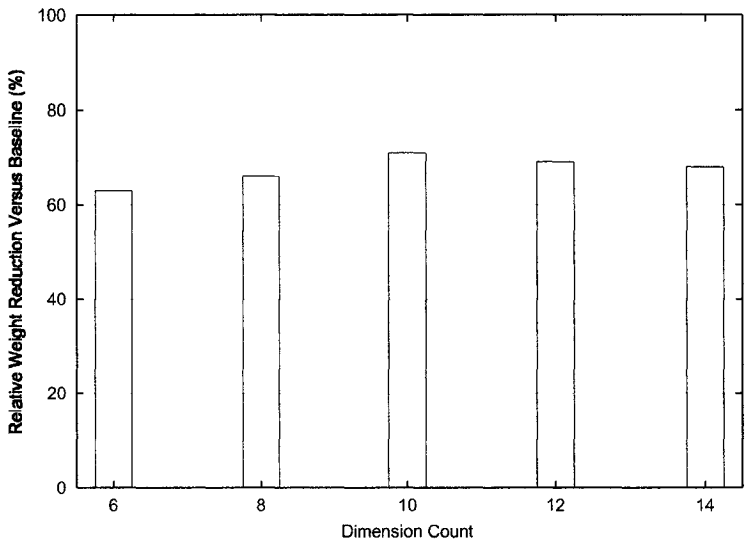


Figure 4.15: Weight reductions for the recursive quadratic time algorithm when the essential set contains views with three or less attributes.

counts up to 14.

4.7.1.4 Addition of Non-Essential Views

The partial cube algorithms each consist of a pair of cooperating processes, one for building the essential schedule tree, the other for adding non-essential views. Interestingly, for randomly selected subsets, cost-effective non-essential views are generally quite rare. In fact, testing has consistently shown that for such subsets, the number of useful non-essential nodes almost never exceeds ten, even for higher dimensions. More importantly, the global reduction in cost tends to be small, typically no more than five percent. In some situations, it might therefore be useful to simply run the BuildEssentialTree algorithm and trade-off a certain degree of optimality for significantly faster execution. Recall that BuildEssentialTree is quadratic on the selected set; its portion of the run-time is consequently quite small.

However, as noted in the previous section, one of the most important cases for partial cube execution is the selection of views in the lower regions of the lattice. Here, it is both possible and productive to add larger numbers of non-essential views.

In particular, though relatively few non-essential views may be added from the lowest levels of the lattice, many intermediate views will be added in the levels immediately above the selected views. In so doing, the algorithm significantly reduces the cost of building the largest selected views, since all of these group-bys would otherwise be computed directly from the raw data set.

Figure 4.16 illustrates the effect of adding non selected views when the selected set consists entirely of views with three or less attributes. Note that, for this test, we have restricted the evaluation to the lattice-derived cubic time algorithm and the recursive quadratic time algorithm. Notice first that the addition of non essential views reduces the cost of the essential spanning tree by an additional 30% to 50%. Moreover these weight reductions are consistently large from 6 to 14 dimensions.

The second point to make is that there is no benefit in using a cubic time greedy algorithm to add non-essential nodes since the cost reductions are virtually identical on common problem sets (again, the cubic time algorithms were not run on high dimension counts). An earlier graph demonstrated that performance for full cube computation was actually better for the quadratic time algorithm. In that case, the BuildEssentialTree method was used to add *all* views to the schedule tree. We can therefore conclude that from the perspective of schedule tree "quality" there is no reason to choose the slower cubic time algorithms.

4.7.1.5 Pruning the Guiding Graph

As we move into higher dimensions, it becomes increasingly important to limit the size of the input set. Our pruning algorithm performs this task by eliminating those views which are unlikely to provide significant benefit to the schedule tree. To permit users to adjust the level of risk that they are will to assume, a *confidence factor* has been added. Our objective in this section is to (a) determine how significantly the guiding graph can be pruned and (b) understand the impact of adjusting the

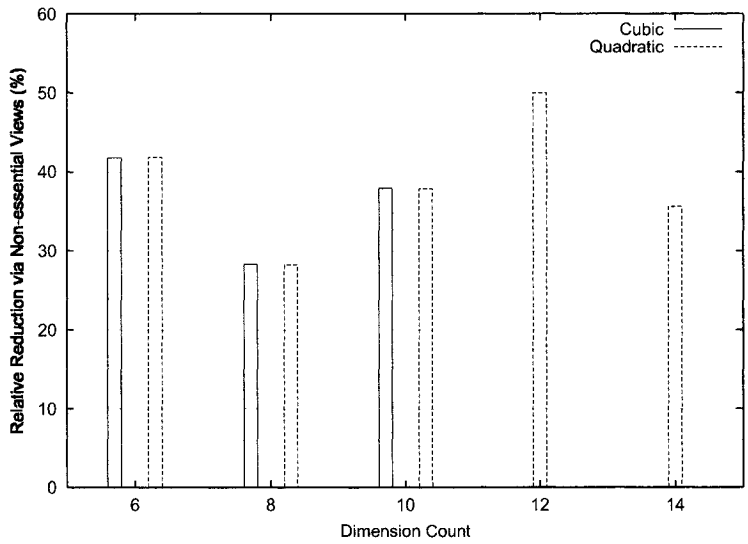


Figure 4.16: The impact of adding non-essential views when the essential set contains views with three or less attributes.

confidence factor.

Figure 4.17 addresses the first issue. As we increase the dimension count — assuming a confidence factor of one — the percentage of views pruned increases steadily from a low of 2% (one of 64 views) at six dimensions to a high of 74% (48,496 of 65,536) at 16 dimensions. Another way of looking at this is that while the ratio of the complete guiding graphs at 6 and 16 dimensions is 1:256, the ratio for the pruned guiding graphs is just 1:67. Not surprisingly, the practical benefit is significant. At 16 dimensions, for example, the pruned input size is just 1/4 the size of the original guiding graph. With an $O(n^2)$ algorithm, this translates into a factor of 16 performance improvement — roughly the same improvement that we would see by running the un-pruned guiding graph on a 16-node parallel machine. Alternatively, we can say that the pruning algorithm allows us to treat a 16 dimension space as though it contained just 14 dimensions.

The second issue we address in this section is the effect of increasing the confidence factor. The default value of one was used for the preceding graph. Figure 4.18 presents results in 14 dimensions for confidence factors from one to three (and views with three

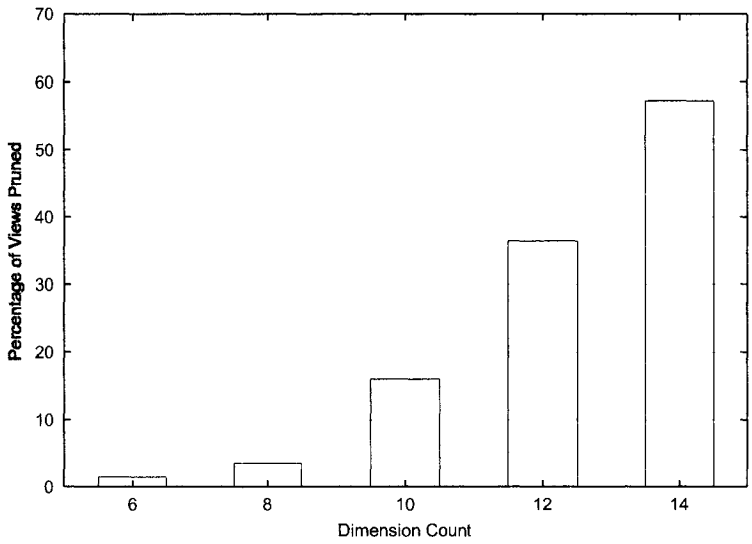


Figure 4.17: Number of views pruned with an increase in dimension count (assuming confidence factor = one).

attributes or less). We note that as the confidence factor increases, there is a huge drop off in the number of views pruned, from 56% to 34% to almost zero when the confidence factor is three. Clearly, a conservative approach to pruning will have a significant impact upon run time performance. More importantly, however, there is virtually no impact upon the quality of the tree as we become more conservative; the relative reduction in cost versus the appropriate baseline is virtually unchanged. As such, we can conclude that aggressive pruning is a low-risk option for improving the run time performance of the partial cube algorithms.

4.7.2 Performance of the Parallel Partial Cube Algorithm

Once a partial cube schedule tree has been computed, it is passed directly to the tree partitioning and pipeline construction modules presented in the previous chapter. In Figure 4.19, we present parallel results on a 14 dimension space in which views of three attributes or less were selected for materialization. The quadratic time recursive algorithm is used for schedule tree construction throughout. Subfigure (a) presents the actual wall clock run time for the complete build, while Subfigure (b)

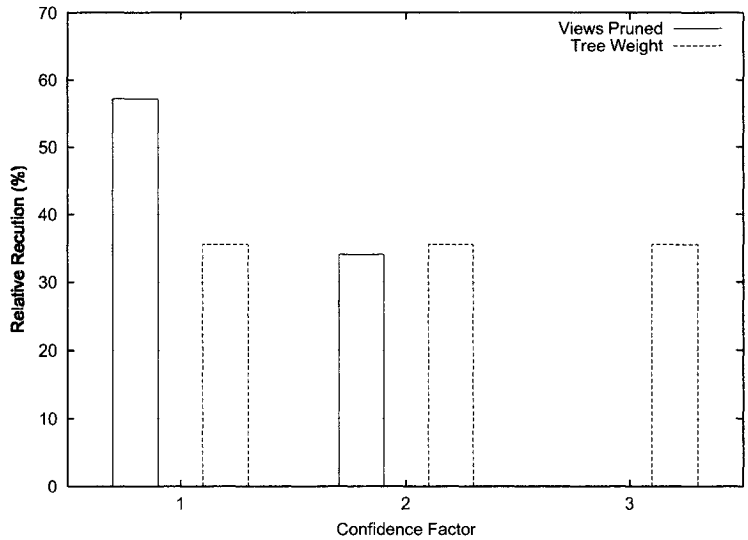
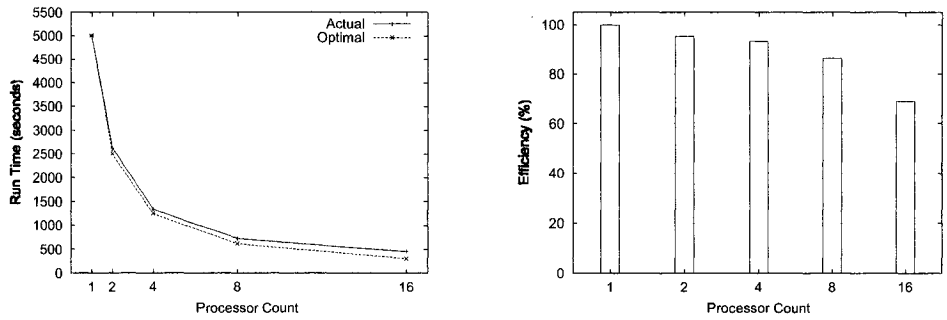


Figure 4.18: The impact upon schedule tree weight reduction as the confidence factor is increased. Note that (i) reduction is relative to the baseline algorithm, and (ii) with a confidence factor = 3, no views are pruned.

presents the efficiency ratings for the same set of experiments. We note that while the general shape of the performance curve in (a) is appropriate, the accompanying efficiency chart shows a rather marked decline to just 68% at 16 processors. Though "acceptable" efficiency measures are somewhat subjective, 68% is quite low relative to our full cube results.

In fact, the decline in efficiency has little to do with sub-par performance by any individual component of the algorithm. Rather, it is indicative of the impact of the larger problem space upon which the scheduling algorithm must run. With a 14 dimension space, the $O(n^2)$ scheduling algorithm took approximately 132 seconds to complete. Furthermore, we note that this time component is constant across all processor counts since the scheduling algorithm runs sequentially on the front-end node. So while the 132 seconds means very little for a sequential build phase that takes almost 5000 seconds, it is a serious performance constraint for a 16-processor test with a 320 second build phase. In Figure 4.20, we present efficiency measurements for the same test, this time with the scheduling component removed. Here, we again



(a) (b)

Figure 4.19: Parallel performance on a data set of 14 dimensions with selected views having three attributes or less. Results for one to 16 processors are plotted as (a) wall clock time in seconds and (b) efficiency ratings.

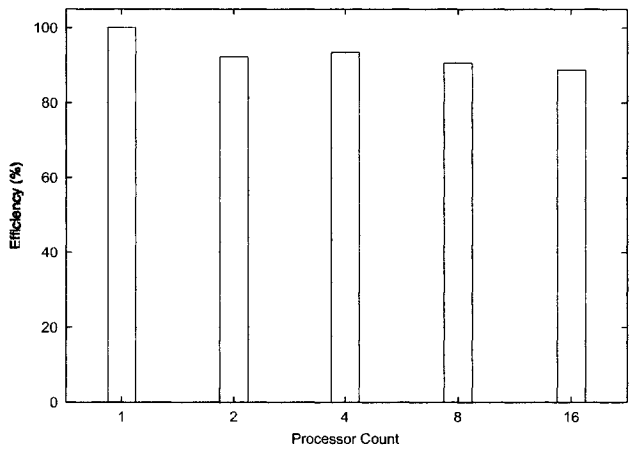


Figure 4.20: Efficiency ratings when considering "build" times only.

see the 90%+ efficiency ratings that were associated with the full cube computation.

Of course, the scheduling component *must* be included within the partial cube framework. Clearly, however, acceptable parallel speedup cannot be achieved if the sequential scheduling component represents a significant portion of the run time on multi-processor implementations. The "obvious" solution here would be to parallelize the scheduling component so that $1/p$ of its execution time was associated with each processor. While this would work, we note that doing so is probably not necessary. In particular, for runs on larger data sets the time taken for the build phase would grow significantly. Conversely, the time taken for the scheduling phase would not

change at all since it is independent of the number of records in the fact table. The efficiency results in Figure 4.20 are therefore likely to represent a better assessment of the algorithm's performance than the results of Figure 4.19(b).

4.8 Review of Research Objectives

In Section 4.3, we identified four objectives for partial cube construction. We now review those goals.

1. **Compute an efficient scheduling tree for the selected cuboid set.** We have provided a suite of algorithms for computing schedule trees for user-specified views subsets. Experimentally, our results demonstrated that the partial cube solutions were competitive with bipartite matching on the full data cube problem and offered significant performance advantages relative to baseline solutions when subsets were required.

2. **The schedule tree should include any non-essential views that reduce the global cost.** Our AddNonEssentialViews method(s) identifies those nodes that are capable of reducing the global cost of the schedule tree. This technique is particularly important for those cases in which the selected views are primarily chosen from the lower levels in the lattice.

3. **The algorithm should be computable/tractable in large spaces.** Though the original model utilized $O(n^3)$ algorithms, we were able to augment the fundamental design so that $O(n^2)$ solutions were possible. These extensions are able to increase the *practical feasibility* of the model by approximately 50%. Moreover, a pruning algorithm was added, further reducing the size of the input set and allowing even larger input sets to be processed. Schedule trees for fact tables with 14 dimensions or more are now easily computed.

4. **The algorithm must be amenable to parallel computing architectures.** By developing new schedule tree algorithms, we were able to plug the resulting graphs directly into the existing parallel partitioning and distribution model. New parallel partial cube algorithms were not required.

4.9 Conclusions

In this chapter we have discussed the design of a suite of algorithms for the construction of partial data cubes. We began with an approach that used "plan" objects to iteratively identify the best candidate views for inclusion within a partial cube schedule tree. Because this method runs in $O(n^3)$ time, which might be prohibitively expensive on larger problems, we then presented algorithmic extensions that allowed schedule trees to be constructed in $O(n^2)$ time, and to be applied to high dimensional spaces. Experimental evaluation confirmed the performance advantage for the quadratic time algorithms, and also suggested than that the Recursive $O(n^2)$ algorithm produced the cheapest schedule trees.

To our knowledge the algorithms presented in this section represent the only such methods for top-down partial cube generation. Recall that while a bottom up method has been proposed [10], it is unlikely to be effective on the dense cuboids often found in partial cube sets. As such, it is likely that our methods represent the most effective partial cube algorithms in the current literature.

In terms of parallel computation, we note that our parallel partial cube method — building upon the partitioning technique presented in Chapter 3 — is the only such solution for ROLAP settings. With respect to the MOLAP context, parallel partial cubes have been presented in [48, 47]. In that case, however, recall that very little attempt had been made to actually reduce the construction costs for the partial cube.

Our experimental evaluation clearly supports the design choices that we have made. Specifically, the combination of the $O(n^2)$ recursive algorithm and heuristic

pruning allows us to efficiently compute partial cubes using parameters and problem sizes that would be meaningful in practical environments. The addition of a supporting parallel infrastructure furthers extends the range of the proposed methods. In short, the research presented in this chapter represents not only an important contribution to the data cube literature, but a powerful solution for real-world problems in the OLAP domain.

Chapter 5

Distributed Data Cube Indexing

5.1 Introduction

In the previous two chapters our primary focus has been the computation of the data cube. Specifically, we have presented new parallel algorithms for the construction of both full and partial cubes. The motivation for generating the cuboids is to subsequently use them to support efficient user-directed queries. While cuboids can be sequentially scanned, in a record by record fashion, the linear time resolution of OLAP queries in this manner would be impractical in the context of production data warehouses given their enormous size. In this chapter we describe a new parallel data cube indexing structure called the RCUBE that supports efficient index construction, bulk updating, and querying in an OLAP context.

In [52], Gupta et al. propose, for the sequential setting, a data cube indexing model composed of a collection of b-trees. While adequate for low-dimensional data cubes, b-trees are inappropriate for higher dimensions in that (a) their performance deteriorates rapidly with increased dimensionality and (b) multiple, redundant attribute orderings are required to support arbitrary user queries. A more interesting option was presented by Roussopoulos et al. [103]. The authors describe the *cubetree*, an indexing model based upon the concept of a *packed* R-tree [102]. The advantage of the sequential cubetree is that it not only provides compact storage and fast query response time, but that it also supports very efficient *bulk incremental* updates, a key

benefit given the size and fluidity of today's data warehouses.

As data warehouses continue to grow in size and complexity, however, so too does the need to explore opportunities for the parallelization of fundamental construction and querying functionality. While significant work has been done on the former — including the work described in this thesis — there has been no attempt to date to provide parallel or distributed algorithms and data structures for relational data cube indexing. With respect to the parallelization of the R-tree — the fundamental cubetree component — a number of researchers have presented solutions for general purpose environments. In [71], Koudas, Faloutsos and Kamel present a *Master R-tree* model that employs a centralized index and a collection of distributed data files. Schnitzer and Leutenegger's *Master-Client R-tree* [107] improves upon the earlier model by partitioning the central index into a smaller master index and a set of associated client indexes. While offering significant performance advantages in generic indexing environments, neither approach is particularly well-suited to OLAP systems. In addition to the sequential bottleneck on the main server node, both utilize partitioning schemes that in the worst case can lead to highly localized searches. In addition, of course, neither approach was designed with OLAP processing in mind and cannot be easily adapted to handle OLAP hierarchies or to support efficient incremental updates.

In this chapter, we describe the RCUBE framework for distributed, high performance data cube indexing in the ROLAP context. Based upon the packed R-tree paradigm, our indexing algorithms and data structures provide load balanced and communication efficient functionality for the construction, maintenance, and access of the relevant cuboid indexes. We also provide a simple but elegant model for the integration of OLAP-specific functionality with the basic retrieval mechanisms. In particular, we address the crucial issues of *partial cube* indexing and the computation of view *hierarchies*. Together, these two features provide the basis of a *virtual data*

cube, a powerful data cube abstraction that provides a rich OLAP interface while hiding the details of the underlying storage mechanisms.

The chapter is organized as follows. We review related work in Section 5.2. In Section 5.3, we formalize the motivation for our work and then, in Section 5.4, we present the algorithmic basis for a new distributed data cube indexing framework called the RCUBE. Section 5.5 discusses the distributed query engine in great detail, including the components that form the core of the virtual data cube model. Extensive experimental results are provided in Section 5.6, while a review of the chapter's objectives are presented in Section 5.7. Final conclusions are provided in Section 5.8.

5.2 Related Work

The fundamental motivation for the data cube is to provide pre-computed summary tables that can be used to efficiently support arbitrary range queries. Referring to Figure 5.1 we might, for example, want to answer queries of the type:

- For the year 1992 provide total sales figure for each car model, broken down by color (three dimensional).
- Provide total sales for white automobiles for the years 1990-1991 (two dimensional).

Though sequential scanning can certainly be used to resolve such queries, it provides acceptable performance only when the query requires the retrieval of a large percentage of the view's records. For fast response time on small to moderate queries, some form of efficient indexing is required. As noted in Section 5.1, b-trees are sometimes used for this purpose. However, in order to support concurrent indexing on more than one attribute, multiple b-trees — one for each attribute — must be created and maintained. Furthermore, the physical layout of records on disk can only be ordered (i.e., sorted) according to one of these indexes, so that while the supporting

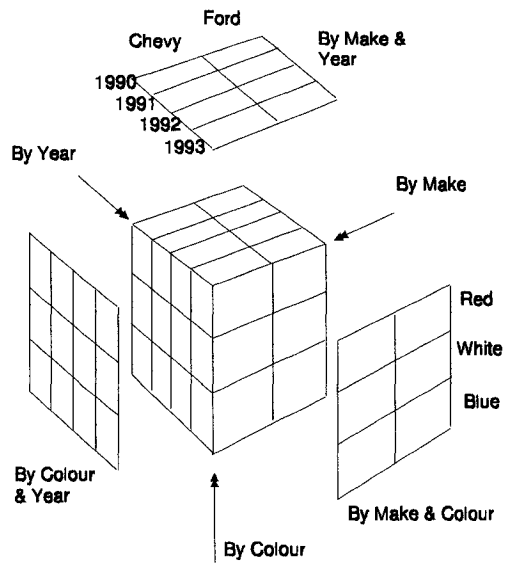


Figure 5.1: A three dimensional data cube depicting automobile sales data.

indexes can be used to identify the records that may belong to the result set (as per the specified dimension), these records will be scattered randomly across the entire disk file. Retrieval performance will therefore be quite poor.

The alternative, of course, is to build upon indexing techniques that have been expressly designed for multi-dimensional spaces. Specifically, we can map the feature attributes of the data cube to the axes of a multi-dimensional grid, while the measure attribute ("total sales" in the previous example) is associated with the cells within the grid. Within the data cube context, it is the R-tree that has attracted the most attention in this regard.

In the remainder of this section we (1) review the sequential R-tree based model for data cube storage and indexing, and (2) discuss techniques for parallelization of the R-tree. While this second group of algorithms is not specifically designed for the data cube problem, the techniques and results described in the associated research

papers are relevant to our own data cube parallelization objectives.

5.2.1 Sequential ROLAP Indexing

In [103], the authors present the *cubetree* model as a means of answering queries on the summary tables of the data cube. R-trees were chosen as the basis of the implementation because (a) a single index supports all dimensions of the view; (b) efficient update algorithms were possible with the packed version of the index; and (c) the R-tree is one of the few multi-dimensional indexes to have been found efficient enough in practice to support commercial products. In the remainder of this section we briefly describe the salient features of the packed R-tree model.

5.2.1.1 The R-tree

As noted in Section C.3, the R-tree is a hierarchical, tree-based index that organizes the query space as a collection of possibly over-lapping hyper-rectangles [53]. The tree is balanced and has a height of $\lceil \log_M |n| \rceil$, where M is the *branching factor* and n is the size of the data set. A query is answered by comparing the values on each of the relevant query attributes with the coordinates of the rectangle that surrounds the points in each data page. The search algorithm descends through successive levels of the index until valid leaf nodes are identified. Because the boxes may sometimes overlap, multiple traversals may be necessary.

5.2.1.2 Packed R-trees

Essentially, the cubetree is a variant of the R-tree that has been *packed* in order to achieve high space utilization and improved query performance. By "packing", we mean that each node of the tree contains as many child references as will fit into a disk block. Though there are a number of different packing algorithms — described in the next section — they all utilize the same general principles.

1. Pre-process the data — usually by sorting — so that the n points are associated

with m pages of size $\lceil \frac{n}{m} \rceil$. The page size is chosen so as to be a multiple of the disk's physical block size. We use the term *Bounding Box* to refer to the space encapsulated by a given page.

2. Associate each of the $\lceil \frac{n}{m} \rceil$ leaf node pages with an ID that will be used as a file offset by parent bounding boxes. Write the pages to disk as consecutive blocks.

3. Construct the remainder of the index by recursively packing the bounding boxes of lower levels until the root node is reached.

Note that packing is only a viable option if the bulk of the data is available "up-front". Since this is true for data warehousing environments, we can use the packed model to dramatically reduce construction times by avoiding point by point insertions. In addition, we guarantee improved storage by fully saturating disk blocks, as well as improved response time by virtue of the fact that points have been pre-processed in a favorable order. Experimental confirmation of these advantages is provided in [102].

5.2.1.3 Packing Algorithms

Central to the packed R-tree is the algorithm used to order data points before they are loaded into individual blocks. Three primary techniques have been developed.

1. lowX [102]: Data is ordered using a conventional multi-dimensional sort.

2. Hilbert-curve [70, 38]: A space filling curve is used to order the data such that points near one another in the original space are more likely to be close to one another in the linearly ordered space.

3. Sort Tile Recursion [107]: The point space is recursively partitioned into tiles or slabs. We note, however, that this method cannot be updated with a linear merge and is thus suitable only for static data sets. Consequently, we will not consider it further in this paper.

The authors of the cubetree chose to employ lowX for their pre-processing algorithm, suggesting that it should outperform Hilbert-based methods on *slice queries*. Slice queries are fairly common in OLAP environments and allow users to view one of the values of a dimension i in terms of a range of values on the remaining $d-1$ attributes. While lowX should indeed perform well on *some* slice queries, Section 5.4.1 demonstrates that it is unlikely to be the best choice for practical index implementations.

5.2.1.4 Packed R-tree Updates

One of the most important features of the cubetree is its ability to efficiently update the original packed R-tree. By exploiting the *bulk incremental update* model that is common in decision support environments, the authors of the cubetree are able to provide an efficient, single pass update mechanism. The update algorithm can be summarized as follows:

1. Collect the "update" data — whose size is typically some fraction of the original input size — and pre-process it in the same manner as that used for the original data. In other words, the algorithm creates a "miniature" version of the data cube using the new data only.

2. Merge the new records into the old set with a simple linear scan. This, of course, must be done for each view independently. In addition, the merged set can be written to fresh storage to allow the existing index/data set to remain online until Step 3 has been completed.

3. Build a fresh R-tree index(s) using the techniques describe above. If the original index/data set has been maintained during the update process, it can now be safely deleted.

Bulk updates avoid record-by-record updates, each of which might warrant an external index/block re-organization. Excessive re-organization limits the amount of CPU time available for query support. In the worst case, dynamic updates can render a large data warehouse almost unusable.

5.2.2 Distributed Relational Indexing

In the parallel setting we can find no previous work on parallel indexes for ROLAP. However, there has been some work on parallel R-tree based indexing. In [69], Faloutsos and Kamel present a parallel R-tree model that employs a single CPU and multiple disks. While appropriate for small indexes, the single CPU approach does not scale well to large OLAP databases. Multi-CPU models are described in [71] and [107]. In the first case Koudas, Faloutsos and Kamel develop a Master R-tree model that places a single R-tree index on the master server and the leaf nodes on auxiliary servers. Designed specifically for a cluster of workstations, the index resolves all *hits* by following pointers to the leaf nodes on the supporting machines. Though more scalable than the single CPU model, the Master R-tree would generate enormous network traffic for the Terabyte-size warehouses in current use. In addition the Master node would become a significant bottleneck in a busy environment.

Schnitzer and Leutenegger improve upon the earlier approach with the Master-Client R-tree model [107]. Here, the global R-tree is augmented with local R-trees on the supporting servers that index their portion of the data. Pointers to remote disk blocks are not required. As a result, almost all of the network traffic is eliminated since the master server only needs to know that at least one leaf node will be found on a given machine. If so, the original query is passed to that server where it is resolved locally.

Since the Master-Client R-tree was designed for a loosely coupled (i.e., general purpose) network of workstations, it was important to only engage a secondary server

if absolutely necessary. The R-tree on the Master node provides this guarantee since it does not contact supporting indexes unless one or more leaf nodes intersect the query space. In real-world high performance OLAP environments, however, it is very likely that the data cube query engine runs on a *dedicated* back-end. In other words, the OLAP server would be the primary application. The Master node therefore creates unnecessary overhead during query resolution. With respect to updates, we note that the Master-Client R-tree model does not support the efficient bulk update mechanisms that would be required in OLAP environments. For these two reasons, performance and maintainability, a more efficient and flexible model is required.

5.3 Motivation

Construction of a distributed OLAP query engine is a complex task, involving not only disk-oriented indexing models, but also the logic for parallel processing and OLAP-centric data representation. In the remainder of this chapter, we present a distributed data cube indexing model that seeks to satisfy the following seven goals:

1. Guarantee the simultaneous involvement of *all* processors in query resolution.
2. Partition the data such that the number of records retrieved per node is as balanced as possible.
3. Minimize the number of disk seeks required in order to retrieve these records.
4. Provide efficient, parallel post-processing functionality.
5. Support the use of partial cubes, such that queries on non-materialized views can be resolved.
6. Support view hierarchies, such that users may query a particular attribute at any available level of granularity.

7. Support efficient bulk updates on the full or partial cubes.

At the conclusion of this chapter, we will return to this list and review the degree to which these objectives have indeed been reached.

5.4 The RCUBE: A New Distributed Data Cube Index Model

In this section, we present a new model for distributed index generation called the RCUBE that not only builds upon existing work in the field but that also adds features uniquely suited to the OLAP environment. To our knowledge, no comparable model has been described in the data cube literature.

The indexing scheme is based on a parallelized R-tree design. Note that the Master-Client R-tree discussed in the previous section consists of a collection of local partial R-tree indexes and a master R-tree that is stored on the front-end and is used to identify which local R-trees must be searched. This hierarchical approach is necessary because it is unclear *a priori* which local subtrees will have data points to report. Our approach, as described in the remainder of this section, is also based on a *forest* of R-trees, but manages to remove the front-end bottleneck and ensure approximately even balancing of the workload through a judicious combination of disk striping and a packing strategy based on space filling curves. The basic model is illustrated in Figure 5.2.

5.4.1 Packing the Data

The authors of the original packed R-tree paper use a technique that is referred to as *lowX* (or sometimes *nearest-X*). lowX is simply a standard multi-dimensional sorting of the view attributes such that for the attribute set A_1, A_2, $A_3 \ldots A_d$, attribute A_1 becomes the *primary* index, A_2 becomes the *secondary* index, and so on. From a performance perspective, the chief disadvantage of this approach is that query response

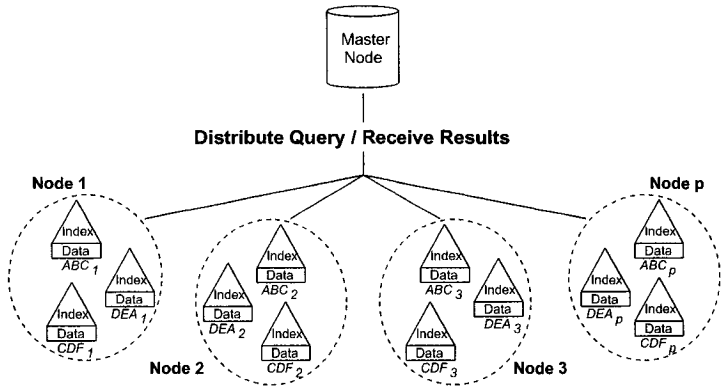


Figure 5.2: The distributed data cube R-tree model.

time deteriorates rapidly when non-primary indexes are required since relevant points are dispersed broadly across the entire data set.

In [70, 38], space-filling curves are used to order data points — prior to R-tree packing — such that values close to one another in the original space are much more likely to be placed into the same disk block (or possibly adjacent blocks). In practice, the Hilbert-curve has been shown to be most effective in this regard and the authors of [70] provide evidence of a significant performance advantage over the lowX algorithm on arbitrary range queries. Figure 5.3 illustrates why. While lowX is likely to work well with slice queries on the X axis, queries favoring the Y axis will touch an unacceptable number of blocks. More specifically, for a d-dimensional view, there are $d!$ possible orderings of the attributes. Only one of these orderings will ideally support a given query; queries with any other order will show varying degrees of deterioration, many far worse than a simple linear scan of the data! In higher dimensions, the problem is only exacerbated. Hilbert-based packing, on the other hand, favors no single dimension and is therefore very well suited to arbitrary range queries.

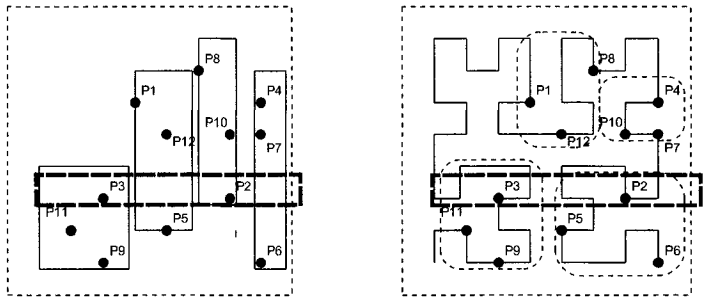


Figure 5.3: Hilbert curve packing versus lowX on a slice query along the "Y" dimension. Note that all blocks intersecting the query rectangle *must* be retrieved.

5.4.2 Data Partitioning

Given that our aim is to balance the retrieval times for arbitrary queries across all p processors, an intelligent partitioning mechanism is a necessity. We note that for dynamically generated R-trees, this problem is very difficult since the insertion of arbitrary records will either destroy the balance or necessitate significant re-balancing costs across nodes. For packed R-trees, however, a much better approach is possible. The fundamental technique is described below.

1. Sort the original data set in Hilbert order.

2. *Stripe* the data across all processors in a round robin fashion such that successive records are sent to the next processor in the sequence. For a network with p processors, a data set of n records, and $0 \leq i \leq p-1$, processor P_i receives records $R_i, R_{p+i}, R_{2p+i}, \ldots R_{\lfloor n/p \rfloor p+i}$ as a single partial set. When $n \bmod p \neq 0$, a subset of processors receives one additional record.

3. For each processor, build the local R-tree partial index from the local striped partition.

The motivation for this striping pattern is that it dramatically increases the likelihood that the space delimited by the hyper-rectangle of an arbitrary user query will

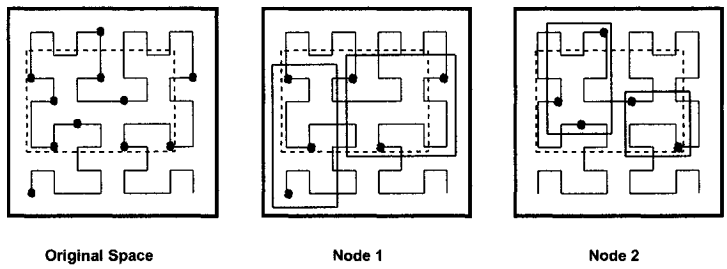


Figure 5.4: Striping the data across two nodes. (Block capacity = 3)

be proportionally distributed across each of the processors in the multi-computer. Figure 5.4 provides a graphical illustration of the intuition. The diagram shows the effect of striping the original space across two nodes. The user query — shown as a dashed rectangle — results in the retrieval of eight points, with each node contributing four points from a pair of contiguous blocks. It is also worth noting that this same example would require four accesses with a sequential implementation of the R-tree, exactly twice the number for a two-node distributed index. The effectiveness of this packing and striping scheme is confirmed by the experimental analysis in Section 5.6.

We note that this method makes no assumptions about the initial distribution of data cube views. While the output of the data cube generation algorithms described in Chapters 3 and 4 consists of a set of complete cuboids on each node, other distribution patterns are possible. For example, a data cube algorithm could generate cuboids individually partitioned across the p nodes. Regardless of the initial location of the cuboids, however, the RCUBE generation process is the same — the data must be striped across the nodes in Hilbert order. In Appendix E, we present the details of the algorithm that is used to build the RCUBE from the output of the data cube generation algorithms described in this thesis.

5.4.3 Updating the Indexes

The parallelization of the RCUBE builds upon (i) the parallel data cube generation algorithm; (ii) the striping mechanism described in the previous section; and (iii)

the sequential update algorithm outlined in Section 5.2.1.4. Algorithm 24 shows how these modules work together.

Algorithm 24 Updating Distributed Indexes

Input: An existing set of indexed views S, distributed across each of p nodes, plus a set of new records U representing the update set.

Output: New indexes that have had U integrated into them.

1: Using the parallel data cube generation algorithm *and* the original schedule tree, construct a *mini-data cube* on U.
2: Using functionality provided in Algorithm 30, stripe the mini-data cube across the p nodes.
3: Using the technique outlined in Section 5.2.1.4, integrate the striped update records into the existing R-trees.

By constructing a mini-cube, we exploit the power of the parallel generation algorithm to efficiently compute summarized records that are ready for integration. We note that the schedule tree for the mini-cube *must* be identical to that of the original schedule tree. This is necessary because the merge in Step 3 requires that the records in the update set and the original set be sorted in exactly the same order. To accomplish this, the original data cube generation algorithm is augmented so that it stores an encoded version of its schedule tree to disk at the time of creation. When the mini-cube algorithm later runs, it recognizes that it is executing in update mode and consequently reads the stored encoding rather than computing its own schedule tree. In so doing, we avoid the possibility that new pipelines, with conflicting sort/attribute orders, could be generated.

Once the scheduling phase has concluded, the appropriate mini-cube is computed and its views are striped across the network. The sequential R-tree update code on each of the p nodes then concurrently merges the records into new collections of indexes, aggregating any duplicates that it finds in the mini-cube "update" partitions.

5.5 The Distributed Query Engine

Previous R-tree parallelization papers [70, 69, 71, 38] have focused exclusively on the retrieval characteristics of the R-tree structures themselves. In an OLAP environment, however, accessing disk blocks is only the first phase of query resolution. More specifically, some form of post-processing is almost certainly required to fully resolve the original query. Two of the most important forms of such processing are (a) partial cube extrapolation and (b) computation of hierarchies. In the first case, the construction of a partial cube implies that some number of views will not be physically materialized on disk. There must be an efficient mechanism for querying against these non-existent views. In the latter case, we note that many attributes can be broken down into a hierarchy of sub-attributes, any of which may actually be more interesting to the user than the parent dimension. Again, the query engine must provide the functionality to address this issue.

In this section, we describe a distributed data cube query engine based on the RCUBE. A general framework for post-processing is presented, along with specific algorithms for handling partial cube indexing and attribute hierarchies.

5.5.1 The Query Engine Model

As noted, the RCUBE has been designed so as to balance the retrieval of query records across all p nodes. Once the records have been obtained, additional OLAP processing is often necessary. The fundamental model — outlined in Algorithm 25 — provides the means by which both forms of computation may be carried out in an efficient, load balanced manner.

In Step 1, the query is broadcast to each of the p processors, thereby avoiding unnecessary bottlenecks on the frontend. The query usually cannot be executed in its *native* form, however, since the user's request is not likely to match the physical ordering of attributes that was determined by the original data cube generation

Algorithm 25 Distributed Query Resolution

Input: A set of indexed views S, striped evenly across p nodes.
Output: Fully resolved query.

1: Pass query Q to each of the p processors.
2: Locate target view T.
3: Transform Q into Q' as per the physical ordering of the records in T.
4: In parallel, each processor j retrieves the record set R_j matching Q'.
5: **for** $i = 1$ to $|R_j|$ on processor j **do**
6: Re-order the attributes of each record i as per the ordering of Q. The transformed records become part of the partial set R'_j.
7: **end for**
8: Perform a parallel sort of R' across each of the p processors. Each processor j now contains a sorted partition SP_j.
9: If required, collect each SP_j into a contiguous buffer on the frontend, ordered SP_1 to SP_p.

algorithm. For example, the user may request a three-dimensional view sorted and presented as A $\times$ B $\times$ C, while the build algorithm may have generated that view as C $\times$ B $\times$ A. In Steps 2 and 3, we identify that view whose dimensions represent a permutation of the dimensions of the user request and then transform the original query so as to match the physical attribute order of the index/view. It is this transformed query that is passed to the query resolution engine. Since the *retrieved* records are not guaranteed to be either ordered (by attribute) or sorted (by record) as per the original query, further processing is almost certainly necessary. In Steps 5-7, the attributes of each record are permuted if necessary during a single linear pass of the partial set. Once this step is completed, a *Parallel Sample Sort* is executed across the parallel machine to produce the sorted order specified by the user. At this point, the view may be returned directly to the user or made available for further OLAP processing. In the former case, a final *Collective Gather* operation pulls the sorted records back into a single buffer. The resulting set is returned directly to the user — no further sequential processing is necessary.

Since there are so many "folk" versions of parallel sorts in the literature, we provide the details of our own Sample Sort implementation in Algorithm 26. It is based upon

the technique originally described in [109]. Informally, a Sample Sort works by having each of the p processors select a collection of p "splitters" from its local data partition. From this splitter set, a single root processor selects a set of p global splitters which best divide the total record space. Using these splitter values, each node partitions its data and sends the subsets to the processors that will be responsible for sorting the relevant portion of the full space. Upon completion of these local sorts, the original data set is fully sorted.

Algorithm 26 Parallel Sample Sort

Input: A k-dimensional set X distributed across p processors, such that for $j \leq 0 \leq p-1$, the partitions $X_0, \ldots, X_{p-1}$ are stored on processors $P_0, \ldots, P_{p-1}$, respectively.

Output: Sets $X_0, \ldots, X_{p-1}$ globally sorted by dimensions $D_{j_1}, \ldots, D_{j_k}$.

1: Each processor P_j locally sorts X_j by $D_{i_1}, \ldots, D_{i_k}$ and selects a set of p *local pivots* (or samples) consisting of the elements with rank 0, $(n/p^2), \ldots ((p-1)n/p^2)$, where $n = |X_j|$. Each processor P_j then sends its local pivots to processor P_0.

2: Processor P_0 sorts the p^2 local pivots received in the previous step. Processor P_0 then selects a set of $p-1$ *global pivots* consisting of the elements with rank $(p + \lfloor p/2 \rfloor), (2p + \lfloor p/2 \rfloor) \ldots ((p-1)p + \lfloor p/2 \rfloor)$ and broadcasts the p *global pivots* to all other processors.

3: Using the $p-1$ *global pivots* received in the previous step, each processor P_j locally partitions X_j (sorted by $D_{i_1}, \ldots, D_{i_k}$ from Step 1) into $p-1$ subsequences $X_j^0 \ldots X_j^{p-1}$.

4: Using one global *all to all* exchange, every processor P_j sends each X_j^i, $i \leq 0 \leq p-1$, to processor P_i.

5: Each processor P_j receiving p sorted sequences X_k^j in the previous step, locally merges those sequences into a single sorted sequence Y_j. The distributed data set is now fully sorted across the p processors.

We note as well that a number of optimizations are included in the current implementation. For example, in Step 4 we retrieve the partial result set and then, in Steps 5-7, re-order the attributes to correspond to the original user query. In fact, the query engine actually combines these steps into a single phase. Specifically, once a disk page has been retrieved into the local application buffer, the query engine must

identify those records within the page that match the query parameters. The matching records are then transferred to a *query result buffer* for further processing. Our query engine performs the record permutation as the transfer is taking place, thereby entirely eliminating the requirement for a separate, and possibly costly, permutation phase.

We also employ a *threshold factor* α to determine whether or not a full parallel sort is required. For very small result sets, a p-node sort would introduce unnecessary communication overhead. If the number of records in the result set is below α, then the partial result sets are sent directly to a single node for sorting. The threshold factor can be tuned to the physical characteristics of the architecture.

In summary, the model represents a fully parallelized data cube query engine. Given an equitable distribution of the result set R across each of the p nodes, the post-processing framework supports re-ordering and sorting operations that are both efficient and load balanced.

5.5.2 The Search Strategy

Much of the previous work in the area of disk-based indexing has focused on the number of blocks retrieved as a metric of index performance [77, 70, 71, 102]. This would appear to be a logical choice since we would expect the increased density of the packed disk blocks to result in fewer disk accesses than would be the case with a conventional R-tree. However, a "raw" count of accessed blocks can be somewhat misleading since the true response time is dependant not just upon the number of accesses but upon the *type* of access. In particular, we must be able to distinguish between (i) blocks that are read following an independent seek — denoted *intra-file* seeks — and (ii) blocks that are read following a read of the previous physical block. We use the term intra-file seek to distinguish *indexing* seeks from *benchmark* seeks. Specifically, the published *average seek time* for a given disk — calculated as $\frac{1}{2}$(number

of tracks per surface - 1) * (track-to-track seek time) — tends to overstate intra-file seek times since (i) blocks from the same file are generally stored on contiguous cylinders and, as such, do no generate long head movements and (ii) the *read ahead* caching mechanism employed on modern disks tends to hide the penalty associated with short seeks. Nevertheless, intra-file seeks can, and do, have a tangible effect upon query resolution performance. For this reason, the simple "blocks retrieved" metric is not an entirely adequate benchmark in our context.

More importantly, however, the scan/seek issue suggests that the standard *Depth First* R-tree search strategy may be poorly suited to the packed R-tree model. Specifically, by searching up and down the R-tree index, we move the read head back and forth along the disk cylinder, thereby increasing the overhead of intra-file seeks. As the size and complexity of the query increases, the growing number of random head movements can lead to some degree of *thrashing*.

Algorithm 27 describes an alternative search strategy that is tailored to the unique structure of the packed R-tree. We refer to this strategy as Linear Breadth First Search. The basic idea with Linear BFS is to visit the same nodes that would be visited by the standard depth first search but to do so in a breadth first manner such that nodes at the same level of the R-tree index are visited in a "left-to-right" fashion. This traversal pattern will correspond to the linear order of blocks on disk.

Before discussing the search algorithm in detail, we briefly review the process of building a packed R-tree. Construction proceeds by recursively packing the $\lceil \frac{n}{m} \rceil$ blocks into a sequence of hierarchical levels, such that all blocks of level i in the tree are constructed before those of $i + 1$ (Note that in this discussion, level 0 represents the root node in the tree). The blocks at each level are then written to disk in reverse order so that the root block is the first page written to disk and the leaf/data nodes are written last. Further, at each level of the tree, the page IDs associated with its k blocks run in consecutive order ID_1, ID_2, $\ldots ID_k$.

Algorithm 27 Linear Breadth First Search

Input: A packed R-tree index, its associated data, and a user query Q.
Output: Fully resolved query.

```
1: Create empty list pageList
2: Initialize pageList with ID of first index block
   {traverse the index}
3: while not at the leaf level do
4:    childList = new empty list
5:    for each page ID i in the pageList do
6:       Using i as an offset, read the relevant disk block B into memory.
7:       for each child block j of B do
8:          if Q intersects the bounding box of j then
9:             Add page ID of j to childList
10:         end if
11:      end for
12:   end for
13:   pageList = childList
14: end while
   {process the data blocks}
15: for each page ID i in the current pageList do
16:    Using i as an offset, read the relevant disk block B into memory.
17:    Process B for records matching Q.
18: end for
```

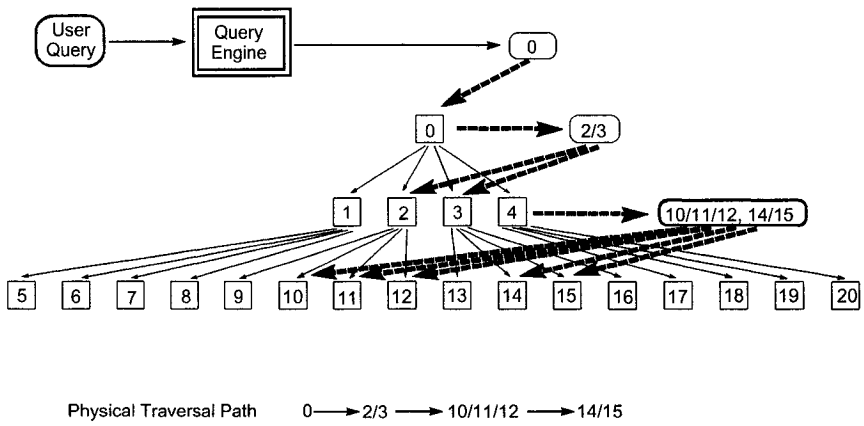


Figure 5.5: Query resolution using Linear BFS. The query is passed to the query engine which, in turn, uses a sequence of page lists to eventually identify relevant records in the leaves/data blocks.

In Algorithm 27, queries are resolved as follows. For the current level i of the tree, the query engine successively identifies the j nodes at level $i+1$ whose bounding boxes intersect the query. It places these page IDs into a *working list*. We note that because of the way the R-trees are built these page IDs are sorted in ascending order. Using the list of j page IDs, the query engine traverses the blocks at level $i+1$ and replaces the current working list with a new list containing the relevant blocks for level $i+2$. It repeats this procedure until it has reached the data blocks at the base of the tree. At this point, the algorithm simply identifies and returns the k-dimensional records encapsulated within the query hyper-rectangle. Figure 5.5 provides an illustration of the application of Linear BFS on a simple query.

The worst case performance of Linear BFS is dramatically superior to the more common Depth First Search strategy on a packed R-tree.

Theorem 6. *The worst case I/O performance of a packed R-tree using Linear BFS is equivalent to*

$$\textit{linear scan of the index} \; + \; \textit{linear scan of the data set}$$

Proof. The construction mechanism of the packed R-tree implies that for an m-page index, the page ID values will run from $0 \ldots m-1$, with the IDs strictly increasing in value in a top-to-bottom/left-to-right fashion. Moreover, the blocks of the data set

also obey this linear ordering. Since the Linear BFS search pattern maps directly to this top-to-bottom/left-to-right physical ordering, it is possible to identify all intersected index/data bounding boxes without ever moving the disk head *backwards*. The worse case I/O performance is therefore equivalent to the time taken to sequentially scan the index, followed by the time taken to sequentially scan the data set.

□

To illustrate the impact of Theorem 6, we issued a query on a pair of ten dimensional data sets. The first contained one million records (43,479 blocks) while the second represented a larger "real world" group-by and consisted of 10,000,000 records (434,783 blocks). The hyper-rectangle of the query was designed to encapsulate the entire space of the set. With a straight sequential scan, the recorded I/O time on the first set was 1.04 seconds and was associated with a single seek to the beginning of the file and 43,478 contiguous scans. Using Linear BFS, the I/O time was 1.17 seconds, with the 12.9% increase tied to the scan of the R-tree index. With a standard Depth First strategy, however, the recorded time was 1.33 seconds, a 28.3% increase over the sequential scan. The additional 0.16 seconds relative to Linear BFS is associated with intra-file seek time generated by 3925 non-contiguous reads. One might expect the results on the larger set to follow a similar pattern. However, it is important to note that the non-contiguous intra-file seeks in very large data warehousing views are much more likely to cross cylinders. As such, intra-file seek time becomes much more of an issue in practice. With the sequential traversal, read time on the larger file was 11.2 seconds. Linear BFS — gracefully degrading to a linear scan of the index and the data file — was once again competitive with an I/O time of 12.8 seconds. The standard Depth First Traversal, on the other hand, generated 39,228 non-contiguous reads and caused the read time to explode to 61.73 seconds, a 550% increase over the sequential scan.

Note that the impact of Linear BFS is related not just to massive queries that are likely to touch all disk blocks, but, more importantly, to the smaller queries that users are most likely to issue. Specifically, it orders the relevant blocks into clusters

of contiguous blocks, each requiring only a single seek and a linear scan. Because the primary strength of the Hilbert space filling curve is its ability to cluster blocks into contiguous chunks, the combination of Hilbert packing and Linear BFS provides tangible reductions in resolution times for most typical queries.

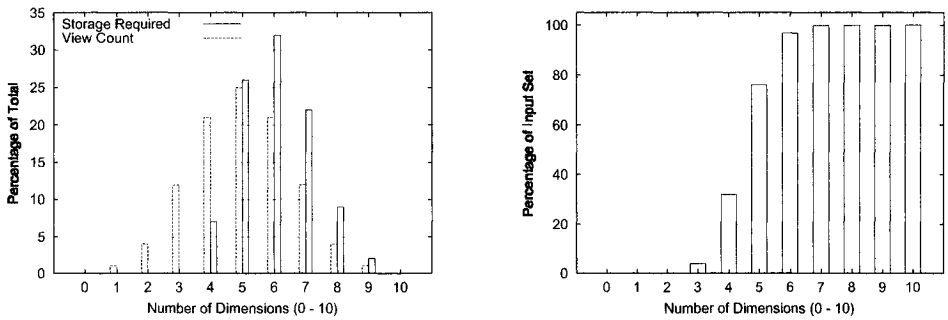
Finally, we should point out that Linear BFS is only beneficial if the logical ordering of disk blocks is equivalent to the physical ordering of blocks. This would not be true in the general R-tree/b-tree case because the dynamic restructuring of the tree due to arbitrary insertions and deletions would quickly permute the original order of the blocks. A Breadth First Search strategy on the logical tree would consequently be no more effective at reducing seek time than a Depth First Search. In the current context, however, this is not the case. For packed data cube R-trees we build the index and data sets in a single phase using all of the values of the underlying data warehouse. We can therefore guarantee that the blocks are written to disk in a completely linear order. Furthermore, because we are using a batch update mechanism for new records, the modified R-trees continue to maintain this same order.

5.5.3 Querying the Partial Cube

As discussed in Chapter 4, it is sometimes desirable to compute just a subset of the 2^d views of the full data cube. Should partial cube generation algorithms be utilized, however, there are two fundamental questions that must be answered. First, by materializing only a portion of the space, how much "information" is lost? And, second, is it possible to efficiently answer queries on non-materialized views? In the remainder of this section, we address these two key issues.

5.5.3.1 An Analysis of Sparsity in High Dimensions

Recall that data cube processing costs are skewed heavily towards the upper portion of the lattice, where the largest views are found. Figure 5.6(a) depicts the relationship



(a) (b)

Figure 5.6: (a) Comparison of view count and storage requirements by dimension. (b) Analysis of record sparsity by dimension.

between the number of views at a given level k of the graph — represented as $\binom{d}{k}$ — and the size of those views for a 10-dimensional data set with mixed cardinality and one million records. Virtually all of the weight is associated with views of five to ten dimensions. At seven dimensions, for example, 12% of the views represent 22% of the weight, while the three-dimensional level has the same number of group-bys but less than one percent of the total weight. However, the skew is even more pronounced than Figure 5.6(a) might suggest since it is actually the number of records within a view rather than the view's physical size which is important. This is the case because the views in higher dimensions must store additional attribute values even if they contain the same number of records as a lower dimensional view, thereby making them only *appear* to contain additional information.

In Figure 5.6(b), we look more closely at the record distribution for the same problem instance. The total number of records in the aforementioned data cube is 642,197,905 — almost 650 times greater than the one million records in the input set. More striking is the effect of sparsity on the average record counts of views at each dimension. We can see that by six dimensions, the average view contains almost 97% of all records in the original input set, while at seven dimensions the ratio approaches 99.9%. In other words, in the upper portion of the lattice almost no aggregation takes

place and the views are virtually identical to one another.

Of course, an increase in data set size also has an effect on record sparsity. One might imagine, then, that very large data sets might produce dense views at much higher levels in the lattice. To address this issue, we have developed a simple model that, for arbitrary fact tables, allows us to approximate the *density threshold*, the point at which the lattice becomes dramatically more sparse. This method is an approximation since the mixed cardinalities and inherent skew of specific data sets produce a density threshold that may, in practice, straddle several levels in the lattice. Nevertheless, it provides a very useful and informative picture of the degree of aggregation in the views of the complete lattice.

The model is based upon the notion of the "average" cardinality C_{avg} of the data cube. We use C_{avg} to then compute the potential space of a "typical" view at level k (i.e., a view with k attributes) of the lattice. Once this *approximate* potential space is defined, it can be passed to the probabilistic size estimator described in Chapter 3, Section 3.6 to compute the expected size of such a view for the given problem instance. With respect to the approximate potential space, it is defined as follows.

Definition 4. *For a d-dimensional data cube with attribute cardinalities of C_1, $C_2 \ldots C_d$, we define the* ***approximate potential space*** *a_k of a view V with k attributes as*

$$\left(\sqrt[d]{\prod_{i=1}^{i=d} C_i} \right)^k$$

Recall that the cardinality set $C_1, C_2 \ldots C_d$ can be used to form S_d, the potential space for the entire data cube. In turn, we can approximate the "average value" within the cardinality set for the full space by taking the d-th root of S_d. We note that the *mean* average of C cannot be used for this purpose since one or more large cardinality values will produce a mean that dramatically over estimates the cardinalities. The k-th power of this approximated cardinality can then be used to estimate the size of the potential space for any view with k attributes.

Now, for an approximated space a_k on a view V with k attributes, we may use the probabilistic estimation model to approximate the number of records r in V as

$$\sum_{i=0} \frac{1}{(a_k - i)/a_k}$$

The significance of this simple analytical model is that its output is a function of all of the relevant data cube parameters: d, k, C, and n. When applied to practical data cube problems, it can be used to predict the density threshold with surprising accuracy. For example, given the parameters of the problem represented by Figure 5.6(b), the model predicts a proportional size of 34% at 4 dimensions, 72% at 5 dimensions, and 95% at 6 dimensions. These predicted values are within 3% of the observed values — quite impressive given that the data set only approximates a uniform distribution.

As previously noted, however, its real significance is that it permits us to assess the impact of data set size and dimension count on the position of the density threshold. Figure 5.7 provides an illustration of these inter-relationships. Specifically, it depicts the density threshold on data sets of 8, 10, 12, and 14 dimensions, ranging in size from 100,000 to 100 million records. We note that for the purposes of this evaluation, we have defined the density threshold very conservatively. Specifically, a view is considered to be sparse only when it contains at least 99% of the records of the original fact table. Given this definition, the graph illustrates two very important points. First, it confirms the assertion that the density threshold does not increase with an increase in dimension count. In other words, for a fact table of a given size and dimensionality between 8 and 14, the density threshold remains constant. Second, for a given dimension count, it takes a massive increase in input size to significantly increase the threshold. In fact, an order of magnitude size increase is required to move the threshold by a single level.

In the context of data cube indexing, an understanding of the density threshold is

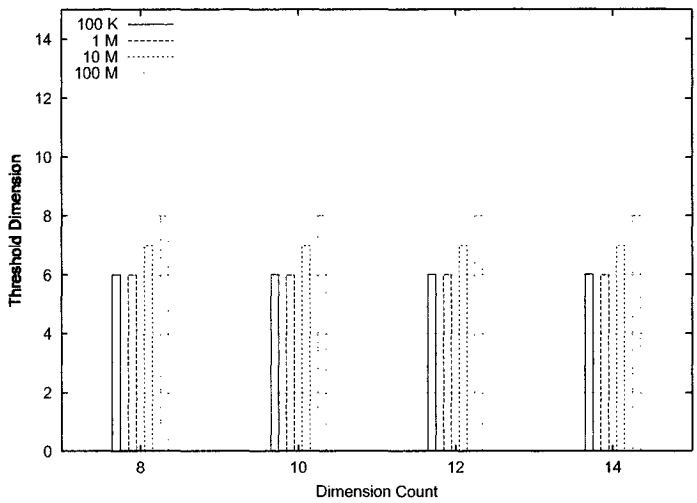


Figure 5.7: The density threshold for varying dimension counts and data set sizes. Note that a data set is considered sparse only when it contains at least 99% of the records in the original fact table.

extremely important. Specifically, it suggests that not only is full cube computation prohibitively expensive but that it may also be almost completely unnecessary since, above a certain point in the lattice, an index on a parent view is likely to be almost as effective as an index on a child view that contains a nearly equivalent record set. In practice, this implies that an effective partial cube will often consist of the *base cuboid* — the d-dimensional group-by that serves as a parent for all views — and some or all of the dense views below the fifth or sixth level of the lattice. The feasibility of partial cube indexing depends of course on the capabilities of the query engine, an issue that is addressed in the following section.

5.5.3.2 The Partial Cube Algorithm

In Section 5.5.3, we discussed the motivation for partial cube generation and indexing. In this section, based upon those observations, we describe how to efficiently answer queries on views in a partial cube when those views have not been materialized. Algorithm 28 describes the extensions to the original model that are required for partial cube query resolution, while Figure 5.8 provides a graphical illustration of the

process.

Algorithm 28 Distributed Partial Cube Query Resolution

Input: A partial set of indexed views S, striped evenly across p nodes.
Output: Fully resolved query.

1: Pass query Q to each of the p processors.
2: Locate surrogate view T.
3: Transform Q into Q' as per (i) the physical ordering of the records in T and (ii) the *peripheral* attributes of T.
4: In parallel, each node j retrieves the records set R_j matching Q'.
5: **for** $i = 1$ to $|R_j|$ on processor j **do**
6: Re-order the attributes of each record i as per the ordering of Q. The transformed records become part of the partial set R'_j.
7: **if** $|T| > |Q|$ **then**
8: Compress each record i as the re-ordering is taking place.
9: **end if**
10: **end for**
11: Perform a parallel sort of R' across each of the p processors. Each node j now contains a sorted partition SP_j.
12: In parallel, each processor aggregates duplicate records that have been introduced by processing the surrogate view.
13: If required, collect each SP_j into a contiguous buffer on the frontend, ordered SP_1 to SP_p.

There are a number of key differences between the new algorithm and the original. First, a *surrogate* view is used as the basis of query resolution. A surrogate is an alternate view that will be used to answer the query on the view requested by the user — termed the *primary* view. To select a surrogate, each node scans its local disk to find those views whose dimensions represent a superset of the dimensions specified by the user. Of the views in this *surrogate pool*, it selects the view of minimum size. We note that since this surrogate view contains even more detailed information than the original view, it can answer *all queries* associated with the original (the reverse, of course, is not true). Furthermore, we note that because Hilbert-based R-tree packing has been used, we do not have to concern ourselves with the physical ordering of the records in the view, since the Hilbert curve does not favor any particular order. As

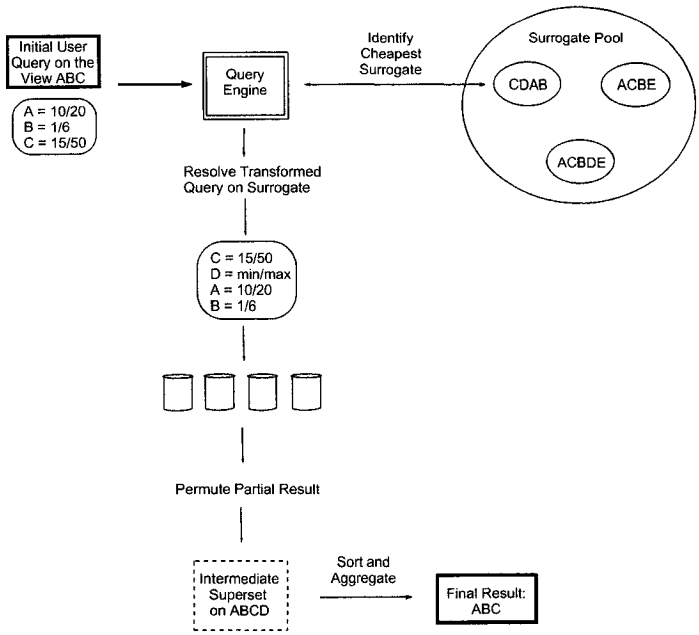


Figure 5.8: The process of resolving a user query on a non-existent view.

noted in [111], if data is sorted according to the more traditional lowX ordering, the only view Q that could be used to efficiently answer queries on an alternative view T is one in which the attributes of Q represent a prefix of T. Since this situation is unlikely to occur often in practice, the resolution of such queries can be quite costly.

Once the surrogate has been determined, the query is transformed by (i) re-arranging the attributes of the query to match the order of the surrogate and (ii) adding the *peripheral* attributes of the surrogate to the the original query. A peripheral attribute is a dimension that is not part of the user query but that must be passed to the query engine in order to resolve the query on the surrogate. In particular, the *low/high* query values for the peripheral dimensions are equated with the minimal and maximal values in the database. In other words, the query engine is told to match records according to just the ranges of the attributes in the original user query; records are never rejected based upon the values of the peripheral dimensions. The end result is a superset of the records that would have been retrieved had

the primary view actually existed. We add that since partial cube indexing is most attractive within environments in which data sparsity creates large views of almost identical size, then the difference between the sizes of the surrogate result and the primary result are likely to be small. In addition, since the goal of the R-tree is to arrange blocks so as to support contiguous retrieval of disk blocks, the time taken to answer the query on a surrogate view will be even less influenced by the use of this alternate view than one might think, as the additional blocks are likely to be accessed with contiguous scans rather than costly seeks.

Once the records have been retrieved, they must be transformed as per the user query. If surrogates are used, the peripheral attributes are dropped from the result set when the re-ordering is performed. The measure attribute is not modified in any way and is retained in the transformed set. The direct elimination of the peripheral attribute is perfectly acceptable since the information it represents is irrelevant to the user query and will not be used again. Moreover, the elimination of the peripheral attributes introduces no additional overhead into the process.

At the conclusion of the parallel sort of the fully transformed partial result set, we extend the original algorithm with an aggregation step. This final piece of processing is necessary because the use of a surrogate may introduce multiple records at the level of granularity represented by Q, even though no such duplication exists within the surrogate. For example, the records $< a_1, b_3, c_3 >$ and $< a_1, b_3, c_4 >$ would be unique within the surrogate view ABC, but would require aggregation into a single $< a_1, b_3 >$ record for the primary view AB. We note that the pseudo-code of Algorithm 28 describes this operation as a distinct $O(n)$ step. In actual fact, the aggregation is integrated directly into the parallel Sample Sort. As was the case with the sorting components of the data cube generation algorithms, we use optimized pointer-based sorting techniques within the Sample Sort framework. Once the partial result set has been "indirectly sorted", it must be physically re-arranged as per the sorted pointer

list. During this $O(n)$ pass, we may compare the record referenced by pointer i with the record referenced by pointer $i+1$. When a duplicate in position $i+1$ is detected, its measure value is simply aggregated into the measure value of record i in the result buffer. As a result, we only require a single pass through the sorted partial set to produce the result buffer, rather than the two passes of a more naive implementation.

The partial cube indexing extensions build directly upon the model designed for standard queries. Given the use of Hilbert-based R-trees, very little overhead is generated by the access phase since any additional blocks from the surrogate are very likely to be retrieved with contiguous seeks. Moreover, the additional OLAP processing — to transform the surrogate query — is primarily associated with a pair of $O(n)$ scans that have been directly integrated into existing processing loops. The end result is a distributed query engine that is almost as efficient on surrogate views as it is on ones that actually exist, as demonstrated experimentally in Section 5.6.

5.5.4 Querying Hierarchical Attributes

In practical data warehousing environments it is often the case that an attribute may be sub-divided into a number of hierarchical levels. For example, the Product Number dimension may be decomposed into a Product Type sub-attribute that, in turn, may be rolled up into a Product Category sub-attribute. While many research papers have simply remained silent on this topic [10, 50, 57, 101, 122], a number of others [110, 105] have proposed treating the sub-attributes as additional "standalone" attributes, in which case the number of views to be materialized would grow dramatically. In [110], the total number of group-bys in the presence of hierarchies is given as $\prod_{i=1}^{k}(h_i + 1)$ when constructed from a data cube with k attributes, each with hierarchies of size h. For example, while a 10-dimensional data cube without hierarchies would generate $2^{10} = 1024$ views, the same data cube with three-level hierarchies on each dimension would produce over one million group-bys. Dealing with hierarchies during the data

cube generation phase is therefore extremely expensive in terms of time and space, even if a partial cube approach is taken. Our approach is to map queries on sub-attributes in an attribute hierarchy to queries on the base or most detailed attribute in the hierarchy. Note that by taking this approach, we are able to exploit the aggregation work performed by the data cube algorithms such that the overhead associated with managing hierarchies is reduced to a small run-time computation phase.

5.5.4.1 Hierarchical Attribute Representation

Contrary to the existing proposals in [110, 105], we address the issue of attribute hierarchies during the query phase by further extending the query model described in Section 5.5.3.2. Before presenting the augmented algorithm, however, we provide some additional background information on attribute hierarchies and their physical representation.

A hierarchy is constructed on top of a *base* attribute $A_{(1)}$, which can be interpreted as the finest level of granularity on that dimension. With our earlier example, the base attribute would be Product Number. The secondary attribute $A_{(2)}$ would be Product Type, while the tertiary attribute $A_{(3)}$ would be Product Category. For a hierarchical attribute A, information captured by the attribute $A_{(i)}$ can always be obtained from $A_{(j)}$ when $i > j \geq 1$. This understanding is fundamental to the model presented in the remainder of this section, in that data will be stored only for the primary attribute. As we will see, queries on other sub-attributes are "mapped" to this granularity level.

We work with both implicit and explicit hierarchies. An *implicit* hierarchy exists irrespective of the database contents. The most obvious example would be the "Time" hierarchy — for example, day, month, quarter. An *explicit* hierarchy, on the other hand, exists only in the context of the current database. An alphanumeric Product

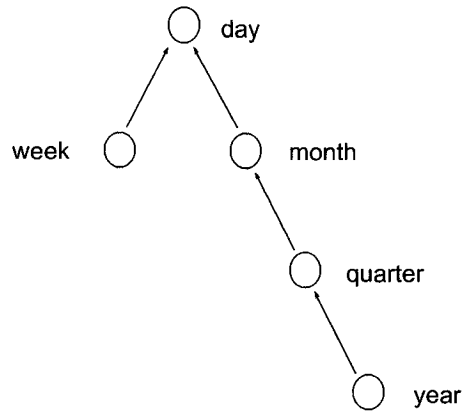


Figure 5.9: The Time hierarchy. Note the two distinct branches.

Number might be one such example since these identifiers are typically defined by the corporate or organizational user. Note that for both explicit and implicit hierarchies the OLAP system requires meta data that describes the structure of the attribute hierarchy.

Figure 5.9 depicts the hierarchical representation of the Time hierarchy. In this case, the primary attribute is "days" — the time hierarchy will be physically represented on disk in this form. Note that "weeks" is contained in its own distinct branch since it cannot be computed in terms of any of the other sub-attributes.

We now describe the notion of hierarchy *linearity*. First, note that $A_{(i)}$ is considered a *direct descendant* of $A_{(j)}$ if $A_{(i)}$ is the child of $A_{(j)}$ in the hierarchy. A hierarchy is linear if for all direct descendants $A_{(j)}$ of $A_{(i)}$ there are $|A_{(j)}| + 1$ values, $x_1 < x_2 \ldots < x_{|A_{(j)}|}$ in the range $1 \ldots |A_{(i)}|$ such that

$$A_{(j)}[k] = \sum_{l=x_k}^{x_{k+1}} A_{(i)}[l]$$

Informally, we can say that if a hierarchy is linear, there is a contiguous range of values $R_{(j)}$ on $A_{(j)}$ that may be aggregated into a contiguous range $R_{(i)}$ on $A_{(i)}$. As a concrete example, the Time hierarchy is linear in that a contiguous range of "day" values — say, 15 to 41 — can always be aggregated into a contiguous range of

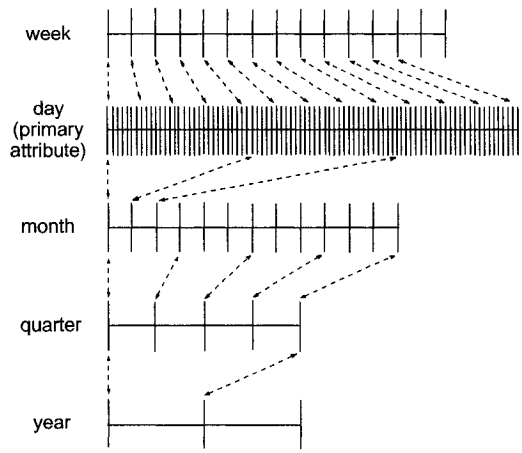


Figure 5.10: The linear relationship of sub-attributes in the Time hierarchy.

"month" values — in this case 1 to 2. Figure 5.10 illustrates the concept of linearity on the Time hierarchy. Note how contiguous ranges of values on sub-attributes always map to ranges of values on their direct descendants.

While it is relatively easy to understand the relationship of sub-attributes within the Time hierarchy, the linearity of other attributes is not always immediately evident. With an alphanumeric Product Number, for example, it is not even clear how a Product Number such as "BY26T7999" compares to one like "GT45J7586" (in terms of $<$ *or* $>$ operations). The process of mapping ranges of Product Category or Product Type sub-attribute values to a corresponding range of Product Number values is therefore not clearly defined.

In the remainder of this section, we describe how query processing is performed on linear attribute hierarchies. To do so, we must actually step back from the indexing problem temporarily to address the problem of OLAP encoding. An attribute *encoding* is the form taken by an attribute value when it is stored in the fact table. Specifically, the format of categorical or discreet attributes is typically converted from its native form in the operational data base into a more compact integer format in the data warehouse fact table. For example, a 64-character Customer string might be

mapped to a 32-bit integer so that "David Witherspoon" becomes Customer "123". Not only does this dramatically decrease storage requirements (from 64 bytes to 4 bytes in this case) but it significantly improves performance for key operations such as conditional checks.

Attribute encoding implies that a mapping system must be employed to allow the query engine to move between encoding models. Specifically, it must be able to translate a user's native attribute value specification A_{nat} into an encoded specification A_{en}. This is accomplished with a *mapping table* that records the correspondence between A_{nat} and A_{en}.

Note that we cannot simply make a linear pass through the native data set and give records IDs simply based upon the order in which they appear. Hierarchical attributes mapped in this manner would be non-linear since an arbitrary mapping at the level of the primary attribute would lead to non-contiguous ranges of non-primary attributes. Instead, we enforce linearity by building mapping tables that are ordered by dimensions $A_{(k)} \times A_{(k-1)} \ldots A_{(1)}$. Figure 5.11 illustrates the mechanism for a three-level Product hierarchy — Product Number (primary), Product Type (secondary), and Product Category (tertiary). The mapping table consists of a set of n records, with n equivalent to the cardinality C of the primary attribute $A_{(1)}$ (i.e., Product Number). That is, for each product number, we create a record containing the Product Number and the corresponding Type and Category. A k-dimensional sort — with primary attribute Category, secondary attribute Type, and tertiary attribute Number — is performed on the n records. Upon completion, we associate the distinct values of each column with consecutive integer IDs.

For each sub-attribute in the hierarchy, we now create an ID Mapping Table. For all attributes $A_{(j)}$ other than the primary attribute (i.e., $A_{(1)}$), we create a mapping table that associates each encoded value A_{en} with a native representation A_{nat} and two min/max ranges. The first min/max range represents the corresponding encoded

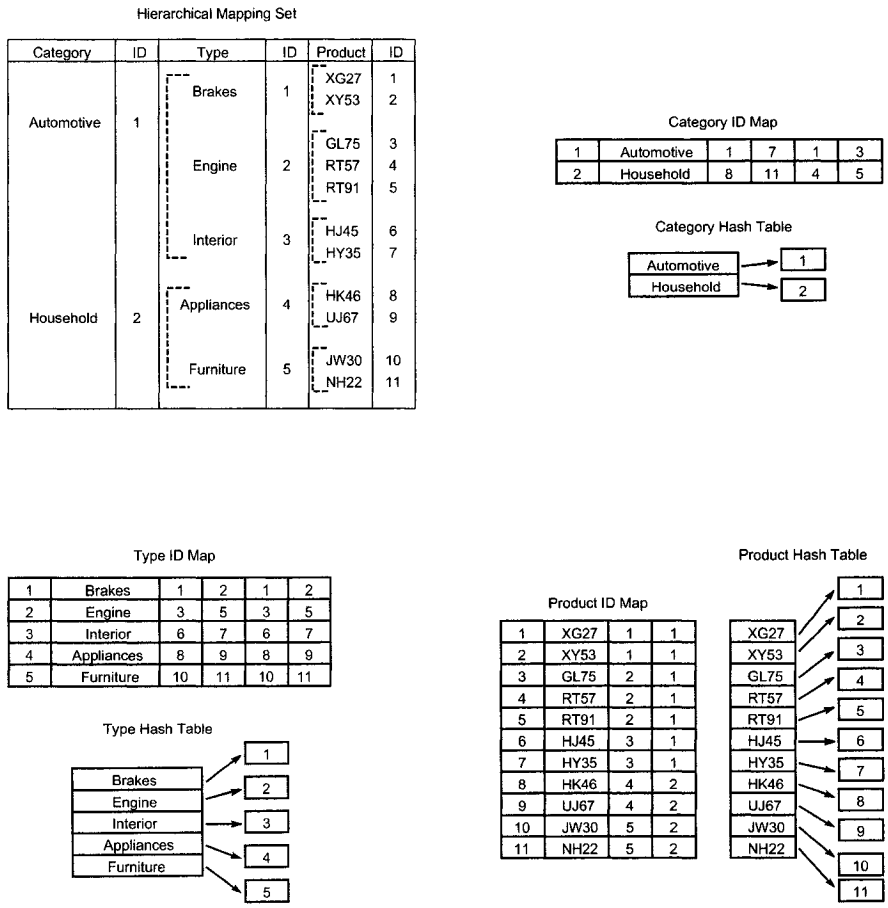


Figure 5.11: The mapping tables.

range in the primary (i.e., physically stored) attribute. The second range corresponds to the encoded range at the next level of the hierarchy. Note that storing both value ranges is not strictly necessary but is likely to improve performance in practice.

For the primary attribute, the mapping table is slightly different. In addition to the native encoding, each record contains $k-1$ *translation fields*, each corresponding to the encoded value of the associated non-primary sub-attributes. We note that for all mapping tables, we get $O(1)$ access time to the records in any other mapping table.

The mapping tables are physically stored on the local disks of each node. When hierarchies are being processed, they are (i) read into memory and (ii) used to create an associated hash table. The hash table is simply used to provide $O(1)$ conversion of native, user-supplied values to encoded system-specific values.

In summary, our native-to-encoded mapping model provides the means by which to define explicit representations of arbitrary hierarchical data cube attributes. Moreover, it guarantees a linear encoding of the hierarchy such that contiguous ranges of values at a granularity level i map to contiguous ranges at granularity level j. In the following section, we will describe exactly how the query engine uses this model to efficiently answer queries on linear attribute hierarchies.

5.5.4.2 An Algorithm for Querying Views with Hierarchical Attributes

The augmentations to Algorithm 25 required for hierarchical attribute processing are presented in Algorithm 29. Figure 5.12 graphically illustrates the process. There are three primary extensions to the original algorithm. The first extension is actually associated with the data cube build algorithm where, as noted in the previous section, we require (a) that hierarchical mapping tables be created and (b) that the hierarchical attribute always be stored *and* indexed in its primary or base form.

Returning to the query algorithm itself, each query defined by the user in terms

Algorithm 29 Distributed Query Resolution for Hierarchical Attributes

Input: A set of indexed views S, striped evenly across p nodes. For views constructed from hierarchical attributes, the base attribute representation is used. Each node also contains a copy of the appropriate mapping tables.

Output: Fully resolved query.

1: Pass query Q to each of the p processors. The query includes a hierarchy level specification for the relevant attribute(s).
2: Locate target view T.
3: Transform Q into Q' as per (i) the physical ordering of the records in T and (ii) the base attribute(s) of T, using the mapping tables.
4: In parallel, each processor j retrieves the record set R_j matching Q' on the primary attribute.
5: **for** $i = 1$ to $|R_j|$ on processor j **do**
6: Re-order the attributes of each record i as per the ordering of Q. The transformed records become part of the partial set R'_j.
7: **if** attribute level A in T != attribute level of A in Q **then**
8: For each record i, use the Primary Mapping Table to translate the relevant value back to the appropriate level of the hierarchy.
9: **end if**
10: **end for**
11: Perform a parallel sort of R' across each of the p processors. Each node j now contains a sorted partition SP_j.
12: In parallel, each processor aggregates duplicate records that have been introduced by processing the base attribute.
13: If necessary, collect each SP_j into a contiguous buffer on the frontend, ordered SP_1 to SP_p.

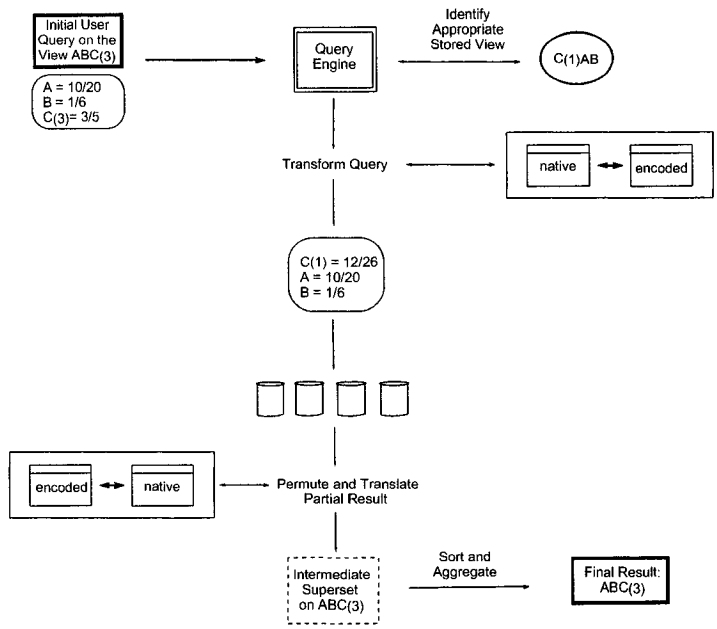


Figure 5.12: The process of resolving a user query containing a hierarchical attribute.

of sub-attributes is transformed into an equivalent query on the primary attribute. A query transformation function is used for this purpose. In turn, the transformation function relies upon the meta data and the mapping tables. The tables presented in Figure 5.11 can be used to demonstrate this process. We will assume that the user has specified a query on the product dimension, but has asked only for products belonging to the "Household" category. The query engine uses the Category Hash Table to determine that the system views "Household" as Category "2". The category value is used as an index into the Category ID Mapping table. There, the Household category maps to the contiguous range 8 – 11 on the primary attribute. This is the min/max range passed to the query resolution module.

Once the records have been retrieved from disk, they must be re-ordered to match the user query as follows. Upon retrieving a disk block, the query module identifies all records intersecting the query rectangle. In our example, this set would include those records containing a value between 8 and 11 in the Product field. The system

then uses the Product ID Map to translate the primary value into the appropriate Category value. Again, we note that Steps 5 – 10 of Algorithm 29 are performed in a single pass of the partial result set, leaving a fully transformed table in the application's result buffer.

Once the mapping/transformation is completed, the records are sorted across the p nodes and the partial sets are aggregated to remove any duplicates that were generated by the hierarchy compression. This phase is identical to that of the surrogate processing algorithm.

Algorithm 29, in combination with our mapping model, provides an effective solution to the problem of data cube representation for views with linear attribute hierarchies. We further note that Algorithm 29 and Algorithm 28 are not mutually exclusive. They can be integrated into a single query engine so that we can, for example, compute hierarchies on views that do not exist on disk. Finally, we add that a fully developed data cube system would also want to cache computed results on sub-attributes to avoid re-computation whenever possible.

5.5.5 The Virtual Data Cube

One of the fundamental principles underlying relational database systems is that the intricacies of file storage and management should be largely transparent to the user. In fact, three of the twelve "Rules" of Relational Database systems described by E.F. Codd [18] refer to transparency. In the current context two of the transparency rules are particularly relevant. Specifically, the relational data model should support (i) *physical data independence* (storage and indexing formats) and (ii) *distribution independence* (the number of processing/storage units).

Note that our data cube model supports these forms of transparency in an OLAP context. The user is not required to understand or even be familiar with the Hilbert-based packed R-tree format of the model or the attribute order of the data sets.

Moreover, the parallel nature of the system is also hidden to the user since the indexing/storage model is managed entirely by the query engine.

More importantly, however, the user does not have to be concerned with the number of views that have actually been materialized (full versus partial cube) or the storage of hierarchical sub-attributes. Instead, they may issue queries under the assumption that *all* views and *all* hierarchies are stored on disk and, furthermore, that the data is available in any attribute order. We refer to this conceptual view as the *Virtual Data Cube*. In fact, this powerful form of data cube transparency is one of the primary contributions of this thesis.

5.6 Experimental Results

We have implemented our distributed data cube indexing prototype using C++, STL (the Standard Template Library), and the MPI communication libraries. As noted, the current design has been targeted specifically at fully distributed, "shared nothing" parallel environments. Extensive evaluation has consequently been carried out on the Linux cluster described in a previous chapter. We note that on this machine communication speed is quite slow in comparison to computation speed. We will shortly be replacing our 100 Megabit interconnect with a 1 Gigabit Ethernet interconnect and expect that this will further improve performance results obtainable on this platform.

In the remainder of this section we discuss the results of a series of tests, each designed to explore a particular feature of the indexing model. All tests are carried out on a 10-dimensional data set of 1,000,000 records that has been indexed and distributed across 16 processors. We note that the testing of practical indexing systems is a particularly difficult undertaking. There are two primary reasons for this. First, the resolution time of a single query is typically very short, making it difficult

to meaningfully interpret comparisons on this scale. Second, modern operating systems are very good at caching recently used disk blocks, with the result being that resolution times can vary dramatically from one run to another.

To counter these problems, we utilize a pair of techniques often employed in the database literature. The first is to perform query tests on *batches* of queries rather than single queries. To this end, we have designed a *query generator* that (i) randomly selects views to search (ii) randomly orders the attributes of that view (iii) randomly generates range values on each of the relevant attributes. The resulting batch list is passed to the query engine as a single "job". In the experiments that follow, batches of 100 queries are used, unless otherwise noted.

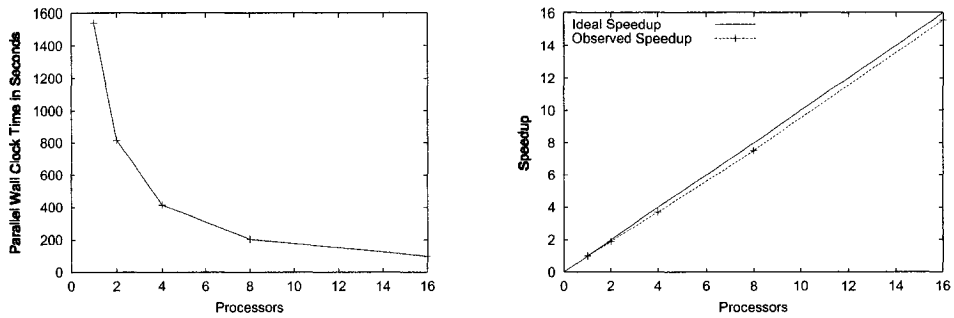
To help minimize the effect of page caching, we *saturate* the page caches on each node prior to every test run. To do so, a number of very large "dummy" files are read into memory. As a consequence, the index/data files of previous runs tend to get expunged from the caches.

The end result is a testing environment that is fair and consistent across all comparative evaluations. Moreover, the results obtained are more representative of real-world, high use systems than would be the case with single query testing.

5.6.1 Index Construction

Figure 5.13(a) shows the parallel wall clock time observed for index construction as a function of the number of processors used. Figure 5.13(b) presents the corresponding speedup.

There are two points to be made. First, from one to 16 processors, our construction method achieves close to optimal speedup — 15.57 on 16 processors. Second, the construction time for 16 processors is just under one minute. In this particular case, the fully materialized data cube consists of ≈640 million rows and 17 Gigabytes of total data.

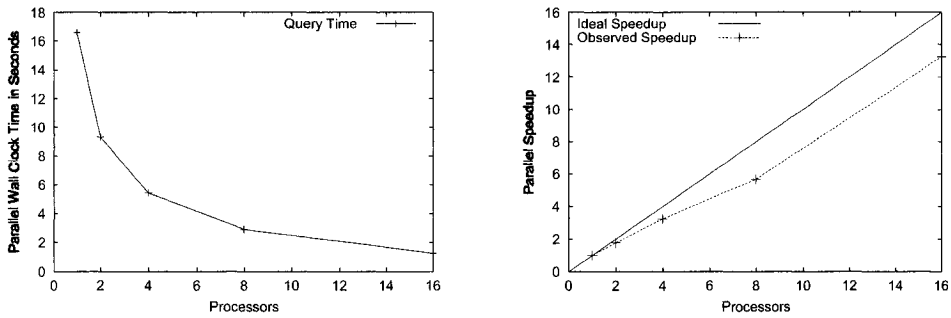


(a) (b)

Figure 5.13: (a) Parallel wall clock time for index construction, and (b) the corresponding Speedup.

5.6.2 Relative Speedup

Figure 5.14(a) shows parallel wall clock time for distributed query resolution as a function of the number of processors used, while Figure 5.14(b) presents the corresponding speedup. We observe that for distributed query resolution the speedup values are quite good. For example, on 16 processors, a speedup of 13.28 is achieved. The source of the difference between this speedup and "perfect" speedup is interesting. Perhaps surprisingly, it does *not* arise from the queries returning different numbers of data points on different processors. As we will see in Section 5.6.4, Hilbert ordering combined with round-robin striping almost perfectly balances the query results over the parallel machine. The small work imbalance observed actually results from the Parallel Sample Sort used to order the query results. Specifically, during its record re-distribution phase, the sorting mechanism does not always partition records as evenly as was the case during the original round-robin striping. This suggests that these speedup results might be further improved by the use of a more balanced sort code.



(a) (b)

Figure 5.14: (a) Wall clock time for distributed query resolution, and (b) the corresponding Speedup.

5.6.3 An Analysis of Scans and Seeks

While wall-clock time serves as a basic measure of performance, it is only a partial representation. Often, the scan and seek counts may be more informative. Previous R-tree papers have tended to treat all block accesses as interchangeable. By simply counting the number of retrieved blocks, however, they at least partially obscure one of the primary benefits of the R-tree. Specifically, not only does the packed R-tree define the order within a block, it defines the order of the blocks themselves. As such, the number of seeks should be dramatically reduced when answering a query on a properly constructed packed R-tree. This is crucial given the significant time difference between a contiguous read and one that requires a seek.

By tagging each block with an incremental ID, we can determine the number of seeks that were actually required to obtain a set of disk blocks. Figure 5.15(a) shows the number of disk blocks retrieved and the corresponding number of disk seeks required in performing distributed query resolution on views of differing sparsity. Note the logarithmic y-axis. Again, we observe the benefit of using Hilbert ordering combined with round-robin striping in our distributed model. Even when a large number of blocks needs to be retrieved, the number of disk seeks across our parallel machine is very small.

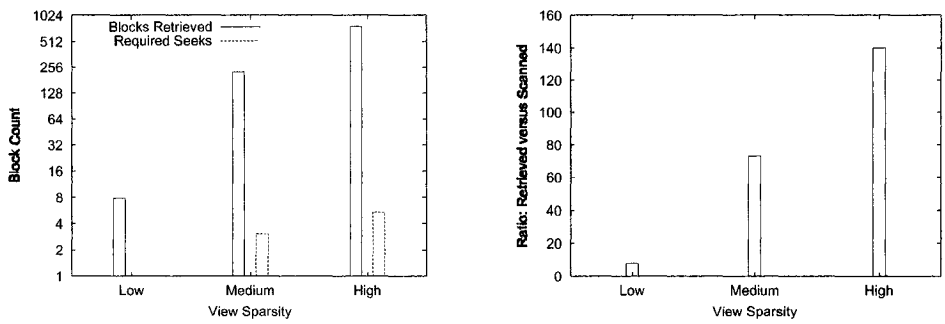


Figure 5.15: (a) Disk blocks received vs. number of disk seeks required on 16 processors, and (b) The ratio of block retrievals to block seeks.

Figure 5.15(b) presents the same information but specifically graphs the ratio between the average number of retrievals and the average number of scans. On large sparse sets, for example, the retrieval/scan ratio is 140 (770 blocks retrieved but only five scans on average).

5.6.4 Retrieval Balance

Figure 5.16 depicts the *relative record imbalance*. That is, for the experiments described in Figure 5.15, we plot the maximum percentage variation between the size of the partial result set returned on each processor. We observe that the Hilbert ordering combined with round-robin striping leads to a maximum imbalance of less than 0.3% with up to 16 processors. In other words, the model produces a record distribution pattern that, for arbitrary range queries, almost perfectly divides the indexing workload between compute nodes.

5.6.5 Hilbert Packing Versus lowX

To demonstrate the superiority of the Hilbert-based R-tree versus the lowX R-tree, we constructed and queried a full data cube using both packing mechanisms. We note that virtually any sorting algorithm can be "plugged in" to the prototype.

Figure 5.17(a) presents a comparison of the number of blocks retrieved using the

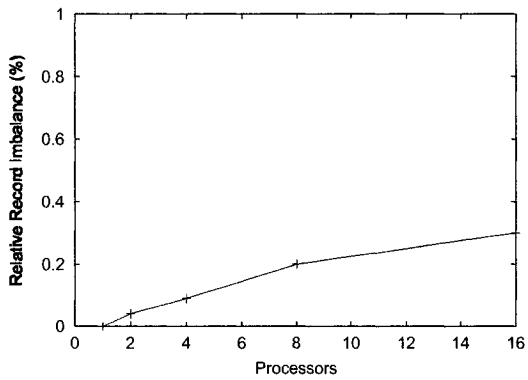
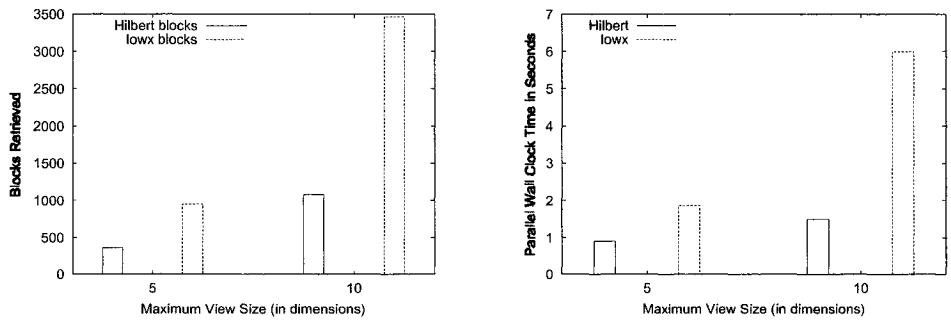


Figure 5.16: The relative imbalance with respect to the number of records retrieved on each node.



(a) (b)

Figure 5.17: (a) Comparison of number of blocks retrieved for Hilbert versus lowX and (b) wall clock *read* time for the same queries

two indexing methods. Query batches were divided into two groups: (i) those containing five attributes or less, and (ii) those drawn from the complete 10-dimensional space. The graph suggests that lowX indexing results in more than three times as many block retrievals on both dense and sparse views.

In Figure 5.17(b), we see a comparison of the *read* or access times for the two alternatives (post processing would be the same for both). For the smaller views, there is a factor of two increase in read time for the lowX indexing method. When we move up to ten dimensions, this penalty increases to a factor of four. We note that in addition to the increase in blocks retrieved, the lowX index generates a larger number of seeks due to reduced block contiguity. In high dimensions, this can produce poor

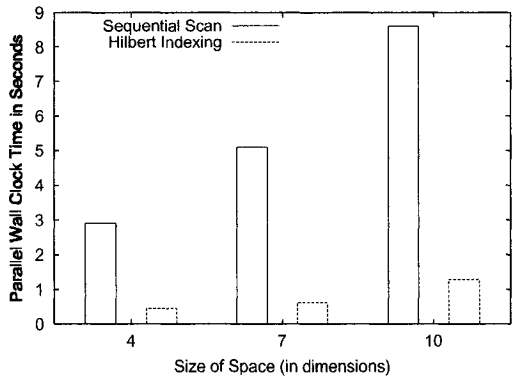


Figure 5.18: Sequential Scans versus Hilbert indexing.

disk access performance.

5.6.6 Indexing Versus Straight Sequential Scans

In the absence of a multi-dimensional indexing mechanism, sequential scans of the data set must be used. In fact, for poorly constructed indexes, sequential scans can often be competitive with respect to performance since they completely eliminate the need for random seeks.

We have designed our prototype with the capacity to retrieve records based upon either indexed access or sequential scans. Figure 5.18 illustrates the performance of the query engine for R-tree indexing and sequential scans for three data cube density levels. Observe that query resolution time for batches of arbitrary queries is between six and nine times faster for indexed views.

We note that as the size of a query relative to the underlying data set increases, there comes a point at which no index can improve upon a sequential scan. In our testing this occurred when the query set exceeded 10% to 15% of all of the records in the data set. However, because of the use of Linear BFS, the penalty associated with unusually large queries is so small that sequential scans would almost never be necessary with our model.

5.6.7 Using Surrogate Views

In this test we evaluate the performance penalty imposed by the use of surrogate views. A query batch was defined and evaluated on a set of existing, materialized views. The materialized views were then deleted and the batch was re-submitted to the query engine so that surrogates were employed on every view. We have, in this case, evaluated the surrogate model by comparing parallel wall clock times for query resolution of primary and surrogate group-bys as a function of the number of processors used. This allows us to see not only the relative effect of employing surrogates, but also any trends that might be related to the distribution of views.

Figure 5.19(a) presents the timings results, while Figure 5.19(b) shows the corresponding relative cost of surrogate-based query resolution over the same search in the materialized primary group-bys. We observe that the overhead of using surrogates is reasonably small, ranging from just over 20% on a single processor to less than 10% on the full 16-node parallel machine. Figure 5.19(b) actually illustrates an interesting trend. As the number of processors grows, the relative cost of using surrogate group-bys decreases. This phenomenon is caused by increased efficiency on the disk accesses when the data is distributed more thinly. Specifically, a smaller partial data set leads to improved clustering of points which, in turn, produces fewer disk head movements.

5.6.8 Querying Hierarchical Attributes

Our prototype has been extended to allow performance testing of hierarchical attributes. Specifically, the prototype can be instructed to treat a particular dimension as the base attribute of the Time hierarchy (i.e., days of the year), but to accept user queries at the secondary or tertiary level. The final results returned to the user will therefore be presented in terms of months or quarters. We note that although the prototype does not currently support the mapping tables required for explicit

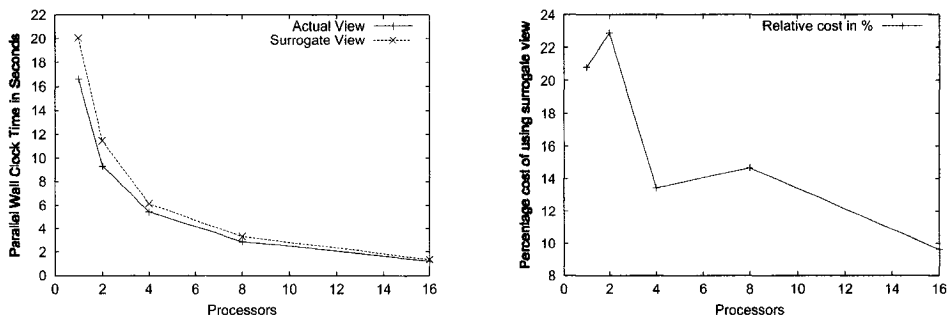


Figure 5.19: (a) Distributed query resolution in surrogate group-bys, and (b) Relative percentage cost of using surrogate view instead of materialized primary view.

hierarchies, much of the computational complexity associated with that form of hierarchical representation is tied directly to the creation and management of the tables themselves. In other words, query performance using explicit hierarchies — and its $O(1)$ mappings — would be quite similar to the results presented in this section.

Parallel wall clock times (16 nodes) for dense and sparse views containing a hierarchical attribute are presented in Figure 5.20. Essentially, the graphs depict the penalty associated with supporting a query on a secondary attribute, as opposed to the same query on the base attribute. Each bar of the histogram is broken down into read time and post processing time ("total" time is the combination of read time and post processing time). To ensure fairness, a batch of random queries specifying the base attribute was first passed to the query engine. This same set of queries was then hand-modified so that the high/low range on the relevant attribute was transformed into an identical range on the secondary attribute. For example, a range of 1/85 (days) would be re-written as 1/3 in the new query set.

The results show a penalty of about 17% for small, dense views, and a penalty of just 6% for larger data cube spaces. In fact, this is the trend one would expect. For views with a small number of dimensions, any additional processing produces a

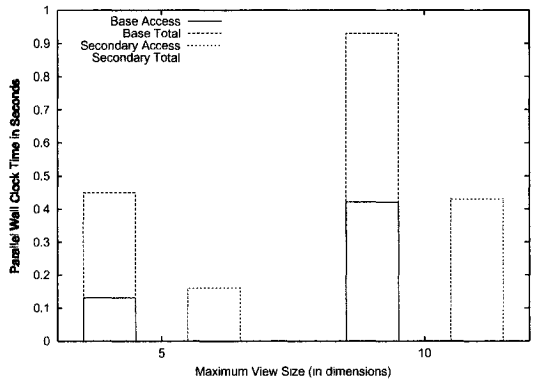


Figure 5.20: Querying performance using hierarchical attributes.

tangible effect. In high dimensions, with much longer disk access time and many attributes to process — both during the record re-ordering and equality checking — the additional cost of a linear number of $O(1)$ mapping operations is relatively insignificant. By extension, of course, we can conclude that the use of multiple hierarchical attributes in the same query would likely create an additional penalty. However, relative to the alternative — computing and storing attributes as separate dimensions — a small run-time penalty on sub-second queries is an enormous advantage.

5.7 Review of Research Objectives

In Section 5.3, we identified a number of objectives for this phase of the research. We now review those goals to confirm the degree to which they have been satisfied.

1. **Guarantee the simultaneous involvement of *all* processors in query resolution.** The model does not generate parallelism by distributing one or more queries to individual nodes, where they would be subsequently resolved in their entirety. While suitable for very high volume query environments (e.g, Web search engines), this kind of approach would provide relatively poor results when the query volume was low to moderate. Our model uses p partial indexes to ensure that all nodes contribute to the resolution of every query.

2. **Partition the data such that the number of records retrieved per node is as balanced as possible.** We combine Hilbert-order sorting and round-robin striping to create our partial indexes. The model produces a distribution of points such that arbitrary query rectangles select an almost identical number of records from each node.

3. **Minimize the number of disk seeks required in order to retrieve these records.** By packing our R-trees with a Hilbert space filling curve, we significantly increase the likelihood that the records of arbitrary range queries are very close to one another on the physical storage device.

4. **Provide efficient, parallel post-processing functionality.** We utilize the same sorting optimizations described in previous chapters when manipulating partial result sets — this time incorporated into an efficient Parallel Sample Sort. Furthermore, $O(n)$ record transformations and aggregation passes are layered on top of other, required operations to further reduce post processing overhead.

5. **Support the use of partial cubes, such that queries on non-materialized views can be resolved.** We utilize a powerful surrogate model for resolving queries on non-existing views. On a parallel machine, the overhead is typically less them 10%.

6. **Support view hierarchies, such that users may query a particular attribute at any available level of granularity.** By storing data at the base attribute level, and providing appropriate mapping functionality, we are able to dynamically transform queries at arbitrary levels of the attribute hierarchy. Overhead for hierarchical attributes is less than 10% for large views.

7. **Support efficient bulk updates on the full or partial cubes.** We combine the core algorithms for data cube generation (full or partial) with an $O(n)$ merge-update operation to allow efficient bulk updates on the indexed virtual data cube.

5.8 Conclusions

In this chapter, we have presented a comprehensive framework for indexing and query processing in a distributed OLAP environment. The model is based upon the R-tree, one of the most successful multi-dimensional disk-based indexing methods. In our case, we have exploited the fact that the data warehouse is available "up-front" to produce packed R-trees that maximize storage capacity and indexing performance. For packing purposes, we build upon the notion of space filling curves that seek to preserve multi-dimensional locality in a single dimension. We use the well-studied Hilbert curve for this purpose.

Since we seek to construct a high performance parallel query engine, the basic model has been parallelized by striping the data across p nodes. Partial indexes are constructed from each partial set. Queries are distributed to each compute node where partial result sets are computed and then sorted and merged to produce a final result.

We have extended the basic model to allow it to support queries on non materialized views, as well as attribute hierarchies. These are two crucial real-world query tasks that are not well studied or supported in the data cube literature. By combining this functionality with that of the original data cube generation algorithms, we introduce the notion of the *Virtual Data Cube*. By this we mean that the user is freed from the requirement of having to understand many of the physical details of the data cube implementation.

Experimentally, our results support the design decisions that we have made. The

system demonstrates good speedup on parallel testing, impressive load balancing, and low computational overhead on surrogate and hierarchy extensions.

In summary, our virtual data cube indexing framework complements the generation algorithms by providing the final piece of an "end-to-end" model for high-performance data cube computing within the ROLAP context.

Chapter 6

Conclusions

6.1 Summary

In this thesis our focus has been the parallelization of the data cube, a data warehousing construct that supports the generation and manipulation of multi-dimensional views of summary data. Specifically, we have addressed the following three issues:

1. **Parallelization of the complete data cube.** Using a coarse grained computing model, we have developed a powerful algorithmic framework for the parallel construction of all 2^d views or group-bys in a d-dimensional space. The fundamental approach has been to distribute the workload between processors by partitioning a weighted schedule tree. We have provided the details of an algorithm for workload partitioning, along with the costing model used to create the underlying schedule tree. In addition we define the issues relevant to high performance pipeline computation and discuss the associated algorithms and data structures. Justification for our design choices is provided by way of analysis and experimentation. In both cases we have demonstrated that, for existing parallel architectures and reasonable problem parameters, our model performs effectively and produces impressive parallel speedup.

2. **Parallelization of the partial data cube.** Contemporary data warehousing systems are often so large that a complete materialization of all 2^d views would

be cost and storage prohibitive. However, the problem of generating partial cubes has not been well studied in the literature. In Chapter 4, we provide a suite of algorithms for the efficient construction of partial cube scheduling trees. After introducing a basic approach to partial cube generation that requires $O(n^3)$ time, we provide refinements of the core algorithms that ultimately lead to new $O(n^2)$ solutions. We further extend this design with a heuristic technique that permits us to address problems in even higher dimensional spaces. By combining the schedule tree generation algorithms with the existing parallel data cube construction framework, we achieve performance and speedup results that are competitive with the bipartite matching "benchmark" on full cubes and vastly superior to more naive approaches on partial cube problems.

3. **Parallel data cube indexing**. Modern OLAP systems require timely access to requested data. Though sequential scanning can be used to resolve query requests, the size of many decision support systems makes such an approach infeasible. We have described a rich parallel model for multi-dimensional OLAP indexing. Our RCUBE index, building upon the widely studied R-tree, uses the Hilbert space filling curve to pre-pack data points into disk blocks. Parallelization is achieved by striping the Hilbert ordered data in a round robin fashion and then generating partial R-tree indexes on the local record subsets. Indexes are then utilized by a parallel query engine. We have extended the core model with support for queries on non-materialized views, as well as views that contain hierarchical dimensions. Extensive testing of the new parallel model demonstrated both impressive speedup on standard queries, and low overhead on surrogate and hierarchy-based processing.

6.2 Future Work

The research described in this thesis represents the core of a robust parallel data cube model. However, the work undertaken to date also points to new research initiatives that would significantly extend the functionality of the current design. Below we identify a number of these possibilities.

- *Automated partial cube specification.* As noted in the previous chapter, one of the goals of DBMS implementors is to hide the physical complexity of the system from end users. While our current model accomplishes this objective in terms of view storage and indexing, it is still necessary for a data base administrator to manually select a cost effective partial cube subset for materialization. This assessment might be based upon such things as the availability and scope of resources as well as the characteristics of the fact table. By supplying an intelligent and efficient algorithmic mechanism for this same purpose, it would be possible to produce a fully automated OLAP environment that selects, constructs, indexes, updates, and queries the data cube without any human intervention. Initial, very preliminary steps in this direction were described by Ullman et al. in [57].

- *Parallel Query Optimization.* Our current indexing model provides efficient support for point and range queries. Of course, not all OLAP queries can be represented by *contiguous* ranges on each dimension. Queries specifying *collections* of contiguous ranges might also be useful. As well, we note that while our indexing model can comfortably handle high volume environments due to the fact that each and every query is processed in parallel, even better performance might be possible by handling multiple queries concurrently. This would involve "piggy-backing" ancillary queries on top of a primary query so that the ancillary query would obtain many of its blocks "for free".

- *OLAP visualization.* Perhaps the most natural means by which to interpret OLAP query results is through some form of multi-dimensional visualization engine. Though commercial OLAP systems sometimes provide such functionality, it often comes by way of "heavy", expensive to maintain, client-side applications and/or inflexible and limited graphical interfaces. It might be interesting to develop a thin, browser-based interface that operates in concert with an extensible server-side OLAP visualization engine.

- *Parallel external memory algorithms.* Recall that our current approach and prototype assumes the existence of an initial input set that can fit entirely into main memory. In practice, on extremely large data sets, this assumption may not be true. To properly support such large initial data sets, it will be necessary to extend the current algorithms — both data cube construction and indexing — into external memory. Although conceptually straightforward, the "systems" work required would be quite significant.

6.3 Final Thoughts

Taken as a whole, the research presented in this thesis supports the notion of what we have called the *Virtual Data Cube*. In keeping with the general philosophy of database management systems, our model provides sophisticated computational functionality in a largely transparent fashion. Moreover, it does so by way of a scalable, parallel, high performance algorithmic framework. Given the importance of the problem itself, both from a commercial and academic perspective, we are confident that the current research represents a significant and meaningful contribution to the data cube literature.

Appendix A

An Introduction to Parallel Computing

A.1 Introduction

Throughout the history of computing, researchers have sought ways of increasing the computing power of existing devices and architectures. Parallel computation has been one of the most obvious means by which to accomplish this. However, the algorithms and computational paradigms in parallel environments demonstrate a much greater diversity than those in the sequential setting. Not only do we have to concern ourselves with more complex hardware, but we must pay considerable attention to such things as cost models, network topology, non-determinism, and data integrity. To further complicate matters, market pressures and technological innovation continue to affect the direction of parallel computing initiatives.

This appendix examines those issues that are most relevant to the current field of parallel computing, particularly as it relates to the development of practical parallel algorithms (for a detailed explanation of these issues, see [14, 15, 24, 72]). Section A.2 begins by examining taxonomies of parallel architectures. In Section A.3, the issue of memory utilization is explored, while Section A.4 reviews the common interconnection fabrics found in parallel environments. Current trends in system design are discussed in Section A.5, focusing on Symmetric Multiprocessors (SMP) designs and Beowulf

clusters. Algorithmic models are presented in Section A.6, with Section A.7 dealing with the issue of performance analysis. In Section A.8, the software currently available for the implementation of parallel algorithms is discussed. Section A.9 concludes with a few final observations.

A.2 A Taxonomy of Parallel Architectures

In 1966, Michael Flynn proposed what has since become the classic taxonomy of parallel computing architectures [43]. Flynn based his classification system on the notion that all computers — both sequential and parallel — could be distinguished in terms of two fundamental elements: the number of data streams and the number of instruction streams. Bearing this distinction in mind, the following four machine classes can be established:

- *Single Instruction Single Data (SISD).* This is the classical von Neumann architecture. It consists of a CPU (control unit and arithmetic-logic unit) and a main memory that holds both program and data. Instructions and data are transferred from memory to high speed registers where they are processed in a sequential fashion.

- *Single Instruction Multiple Data (SIMD).* In many cases, it is possible to perform common operations on distinct blocks of data. This basic observation led to the development of SIMD machines. SIMD architectures use a single control unit to dispatch instructions to a collection of independent processing units. The instructions are then executed concurrently by each of the units (note: some may actually be idle during a given iteration). For applications with a very regular structure, SIMD machines can be quite effective. Prominent examples of this class include MasPar's MP-1 [81] and MP-2 [82].

- *Multiple Instruction Single Data (MISD).* Though this design is technically viable, practical systems are unlikely to be built this way. It is included only for completeness.

- *Multiple Instruction Multiple Data (MIMD).* In this final class, we find those machines that allow independent instruction streams — executed on independent processors — to manipulate distinct blocks of data (see Figure A.1 for an illustration of the difference between MIMD and the aforementioned SIMD machines). We must note here that "independent instruction stream" does not necessarily imply that the instructions themselves are unique. More often, a single program is executed on each processor, a model known as Single Program Multiple Data or SPMD. In any case, MIMD architectures exploit processors that utilize their own control unit and processing element. Resources may either be under the control of a single operating system or may be managed concurrently by independent copies of the operating system. Examples of parallel machines in this class include the Paragon from Intel [92], the T3E from Cray [117], the SP2 from IBM [114], the Power Challenge from SGI [108], and the new workstation clusters [95].

During the 1960s and 1970s, it was the SIMD model that came to dominate the field of parallel computing. Scientific applications, commonly associated with highly structured array-based computing, were ideally suited to the SIMD's tightly coupled, common control unit design. Specifically, parallelism could by achieved by the compiler, rather than by sophisticated coding. In practice, the individual PE elements of Figure A.1(a) — each with their own local memories — eventually gave way to the vector processor. While still an SIMD machine, the vector design consisted of a powerful scalar processor, a vector of associated function units, and a *common* memory. Though expensive, the model can provide tremendous performance

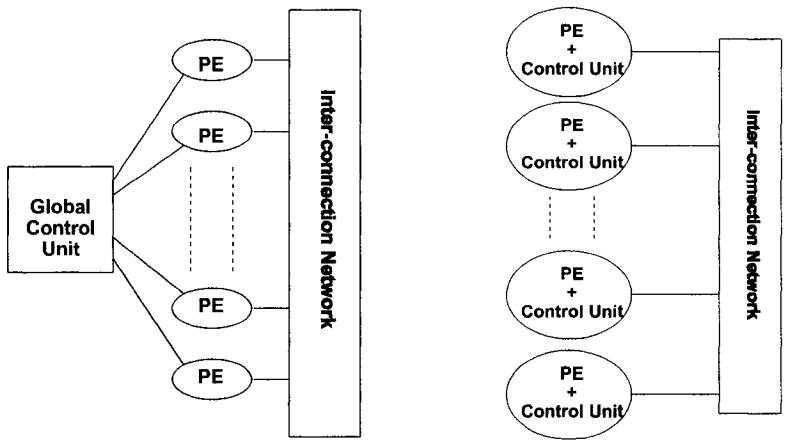


Figure A.1: (a) SIMD Architecture. (b) MIMD Architecture.

on certain application classes. In fact, the term "supercomputer" has historically been used in reference to the vector design. The Hitachi S3600 [60] is an example of the vector model.

While vector based processors are still used for massive scientific applications, the rise of the microprocessor eventually produced more varied and more flexible designs — and a whole new generation of multi-processor systems. By the mid-1990s these "massively parallel processors" (MPPs) were dominating the world of general purpose parallel computing. In fact, in the most recent listing of the *TOP 500* — a ranking of the world's fastest computer systems — only 7.4% of the machines had a vector design [120]. Given the potential range of parallel applications, and the continued investment in commodity processors, this trend is unlikely to change in the near term.

In the remainder of this chapter, we will discuss the MIMD model in greater detail. It is worth noting that despite the ubiquity of the MIMD design, there is in fact great diversity with respect to implementation [72, 73, 76, 95, 24, 14]. In the current context, emphasis will be placed upon those features, both physical and conceptual, that are most relevant to those designing efficient algorithms for parallel applications.

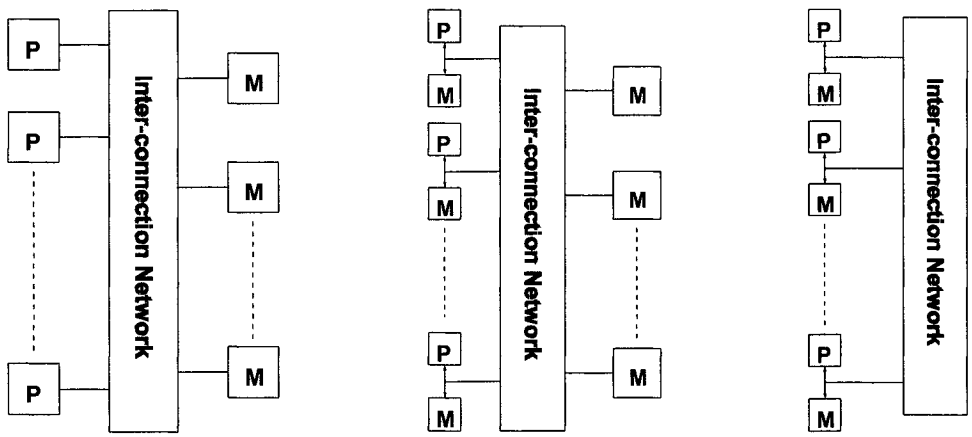


Figure A.2: (a) All memory is global. (b) Processors have a mix of local and global memory. (c) All memory is local. Hardware provides remote memory access.

A.3 The Memory Model

One of the most fundamental distinctions between the sub-classes of the MIMD model relates to how core (i.e., primary) memory is structured and how that memory is accessed by processing units. In particular, there are two very different approaches to the organization of the address space when multiple instruction streams are employed, namely shared memory and distributed memory.

A.3.1 Shared Memory MIMD

With a shared memory parallel design, all processors have equal access to at least some segment of shared memory. These types of machines are sometimes known as *multi-processors* (Note: this term is often applied to parallel machines in general). Figure A.2 illustrates the three basic shared memory configurations:

1. One single pool of shared memory (Figure A.2(a)).

2. A mix of shared global memory and local processor memories (Figure A.2(b)).

3. Local memories only (Figure A.2(c)). It is important to note that in this case

the location of data is transparent to the programmer. In other words, the hardware is responsible for resolving non-local references.

When all memory is global, it is very likely that access times are consistent across processors. Such machines are known as Uniform Memory Access (UMA) computers. Most current shared memory machines, however, are actually Non-uniform Memory Access (NUMA) computers (as in Item 2 and Item 3 in the previous list). On such machines, there may be significant differences in access times for local versus remote memories. A common example is the SGI Power Challenge [108].

To effectively support the shared memory interface, vendors must provide significant hardware support (whether UMA or NUMA). For example, with all processors having equal access to shared data, it becomes necessary to arbitrate access to that data when concurrent reads and writes are encountered. Furthermore, in an attempt to increase processor memory bandwidth, most shared memory machines provide some form of local cache. As a result, cache coherency protocols are required to ensure that the local processors do not employ local variables that are no longer consistent with the global versions.

A.3.2 Distributed Memory MIMD

In contrast to the shared memory computer, a distributed memory system, or *multicomputer*, provides local storage only and may offer no mechanisms for directly accessing the memory associated with remote processors. Instead, data is typically shared by means of explicit *message passing*. In other words, communication software is provided so that the programmer can both send and receive packets of data. Figure A.3 illustrates the generic message passing model.

While message passing machines require less sophisticated hardware and offer the programmer tremendous control over the application's execution profile, they can also

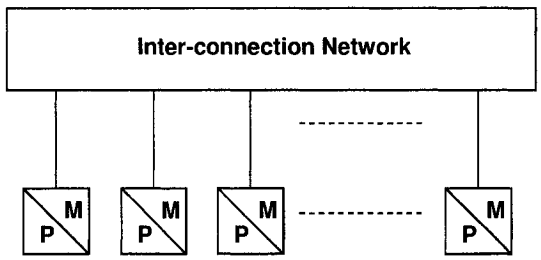


Figure A.3: Each node in a distributed memory parallel machine contains its own CPU and local memory store.

be exceedingly difficult to program efficiently. Because of the significant time difference between local and global access, it is quite possible that badly written programs — or poorly designed algorithms — will result in unacceptable communication delays. Consequently, considerable effort is expended upon both reducing the number and size of communication rounds and hiding communication latency by ensuring that the local CPU has useful work to do during the time it takes for data to travel between processing elements.

A.4 The Interconnection Fabric

Because of the way memory is employed in the two types of MIMD machines, it is perhaps not surprising that the means by which that memory is accessed can also be dramatically different. In this section, we look at the two interconnection architectures, with a particular emphasis on the more varied message passing environments.

A.4.1 Dynamic Interconnection Networks

On shared memory platforms, each processor must be given the opportunity to access some or all of the global memory store. Consequently, the emphasis is on incorporating some measure of redundancy into the processor/memory channels such that a given processor can easily find an "open" memory bank. In other words, should one processor be accessing a given bank, another processor should not be prevented or

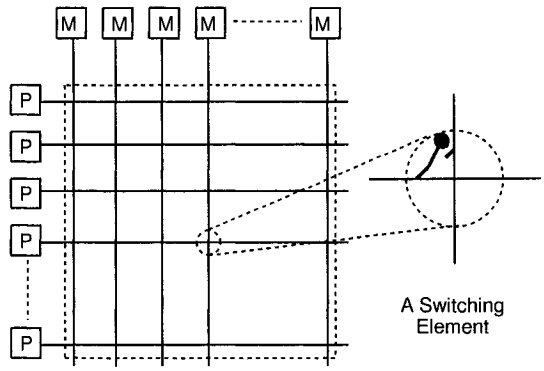


Figure A.4: A crossbar switch.

blocked from dynamically reaching an open bank.

A common means of achieving this is through a *crossbar* switch (see Figure A.4). With a crossbar, a grid of switching elements allows each of p processors access to any of b memory banks, where b is assumed to be at least as large as p (otherwise, some processors would not have access to any banks). Unfortunately, this $\Omega(p^2)$ crossbar grid can be extremely expensive in practice, even for a network with just a few dozen processors.

The prohibitive cost of the crossbar switch eventually led to the design of the multi-stage network. Here, we trade off performance against cost by constructing a series of intermediate switching elements between memory and processors. Figure A.5(a) illustrates the basic design. The network consists of a number of levels, and data transfer is accomplished by routing bytes/words across the fabric.

Finally, at the opposite end of the price performance spectrum is the bus-based access model. See Figure A.5(b). In this case, a single memory bus is shared between each of the processors in the system. While the design is relatively simple, and thus inexpensive, total bandwidth is fixed. As such, bus-based designs do not scale to very large numbers of processors since bus saturation will prevent full utilization of processing resources.

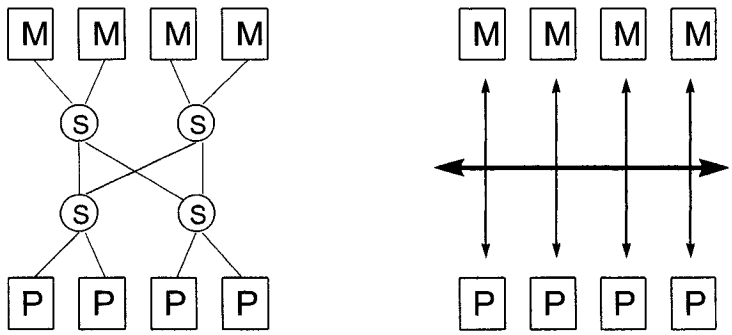


Figure A.5: (a) A multi-stage shared memory network (The "S" nodes indicate dedicated switching units). (b) Shared memory with a common bus.

A.4.2 Static Interconnection Networks

In contrast, message passing architectures typically employ a static interconnection fabric consisting of some number of hard-wired point-to-point links. The following list describes some of the traditional fabric designs:

- In the simplest case, a static network can simply be a *linear array* of nodes connected by a sequence of $p - 1$ lines. By connecting the first and last nodes, the array is transformed into the slightly more powerful *ring* model (see Figure A.6 for an illustration of the array and ring). Not surprisingly, these two simple designs provide relatively poor communication characteristics when non-contiguous nodes need to share data since the message may have to traverse many links.

- A somewhat more compelling network option is the *star* network topology. With a star, a single node acts as a central hub, routing packets between any two points. This design provides clean point-to-point access but does increase the risk of failure due to its single *point of failure*. Note that the central node in the star is a standard compute node, not a dedicated switching element. A more sophisticated version of the star network is the *fat-tree*. It utilizes a hierarchical

(a) (b)

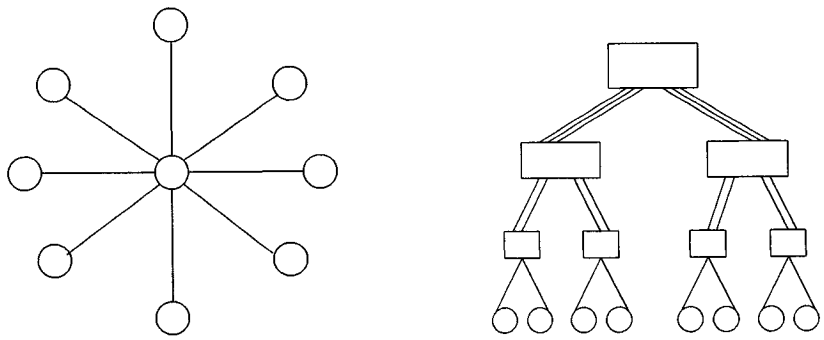
Figure A.6: (a) A simple array. (b) A ring formed by joining the first and last node of the array.

switching fabric and, in so doing, eliminates the single point of failure. As well, the broad bandwidth provided at the top of the tree reduces the bottlenecks that can be created in the simple star network. See Figure A.7 for an illustration of the two designs.

- Another popular interconnection fabric for distributed memory machines is the *mesh*. A multi-dimensional extension of the array, the mesh provides every node with two connections in each of the d dimensions (fewer, of course, at the edges). By connecting nodes on the periphery of the mesh, we obtain what is called a *torus*, the multi-dimensional counterpart of the ring (see Figure A.8).

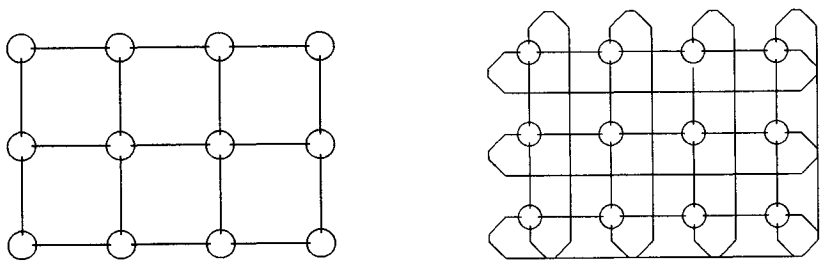
- When even higher levels of connectivity are required, the *hypercube* is often the network of choice. The hypercube is a multidimensional grid of processors with exactly two processors in each dimension. In other words, a d-dimensional hypercube is made up of a total of 2^d processing units. In the general case, a $(d+1)$-dimensional hypercube can be constructed recursively by connecting the corresponding processors of a pair of d-dimensional hypercubes. Figure A.9 provides a comparison between the hypercube and the *fully connected* design. It should be noted that although the fully connected network offers complete node to node access, it is generally too expensive to be used in practice.

As noted above, not all interconnection fabrics are equally attractive. In general, we trade off cost for greater connectivity and stability. In this respect a number of metrics have been proposed which more precisely define the power and complexity



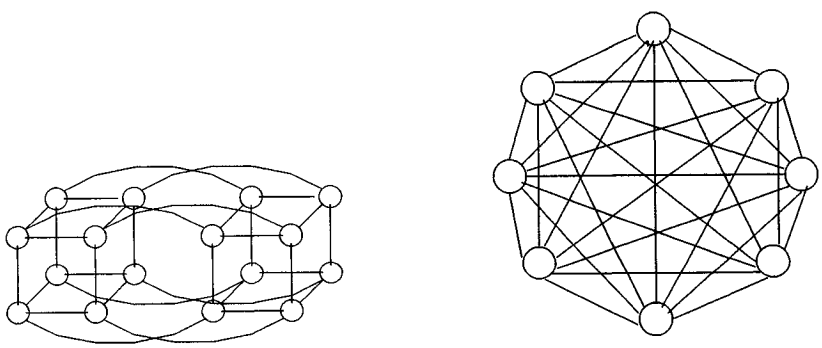
(a) (b)

Figure A.7: (a) The star design. (b) The more sophisticated fat-tree.



(a) (b)

Figure A.8: (a) A four-by-four mesh. (b) The wraparound mesh or torus.



(a) (b)

Figure A.9: (a) A four-dimensional hypercube. (b) The fully connected network.

of a given fabric. The following list describes some of the more important (with examples chosen from the previous list):

- *Diameter.* If we define the distance between any two nodes as the smallest number of links between them, then the diameter is simply the maximum distance between any two nodes. For example, the diameter of a star network is two, while that of a wraparound mesh is $2\lfloor\sqrt{p}/2\rfloor$. Ideally, we would like the diameter to be as small as possible since this reduces communication time between any two nodes.

- *Arc Connectivity.* Connectivity is a measurement of the number of paths between any two processors. By arc connectivity, then, we mean the minimum number of links that must be removed so that that network is divided into two independent networks. In this case, we would like the connectivity value to be as high as possible since this increases both reliability and communication concurrency. The arc connectivity of a completely connected network for example is $p - 1$, while a hypercube has connectivity of $\log p$.

- *Bisection Bandwidth.* Bisection bandwidth is defined as the product of *bisection width* and *channel bandwidth.* By bisection width we mean the number of links that have to be removed in order to partition the network into two *equal* halves. For example, a fully connected network would have a bisection width of $p^2/4$, while a ring would have a bandwidth of two. Channel bandwidth, on the other hand, is essentially a measurement of the peak rate at which bits can be transferred between a pair of network nodes. So bisection bandwidth, then, is a measure of the minimum volume of data that can be communicated between any two halves of the connection fabric. For algorithms that require *global* operations (i.e., all nodes communicating simultaneously), this is particularly important since it places an upper limit on the speed at which the

communication round can be completed.

- *Cost.* Though a number of cost measures can be used, one of the most common is simply the link count. The link count of a completely connected network is $p(p-1)/2$, for example, while a two-dimensional mesh without wraparound has a link count of $2(p-\sqrt{p})$. Ultimately, we can only adjust the first three measures if the cost of adding the required links does not become prohibitive.

A.5 Contemporary Trends

As noted in Section A.2, there has been a general movement over the past 15 years towards the more flexible MIMD machines. Even here, however, the state of the art continues to evolve at a rapid rate. This section examines current trends in MIMD system design.

A.5.1 The Symmetric Multi-Processor

At present the most common *multi-programming* platform is the commodity-based symmetric multi-processor or SMP. In the broadest sense, an SMP is any system that allows the OS to schedule any available process or thread on any of its identical CPUs. A more precise definition, however, is that an SMP is a shared memory multi-processor in which the cost of accessing any element or bank of memory is the same for all processing units. Today, the most common form of SMP is the shared bus, single board systems constructed from cheap but powerful micro-processors, though larger solutions also exist.

We have referred to these machines as multi-programming systems to emphasize the fact that they are also very popular for non-parallel codes. For example, they can be used to concurrently run sequential programs or to exploit multi-threaded servers. Their cost, coupled with the relatively simple interface they present to users, makes them well-suited to this role. It should be noted of course that improved sequential

performance can only be obtained if there are many programs available. A single sequential code runs no faster on an SMP since the OS simply loads and runs it on a single processor.

In terms of parallel computing, the SMP is also quite popular. For small, neatly structured problems, relatively simple parallel codes can produce impressive reductions in run-time on these shared memory, single OS platforms. For larger problems requiring much greater computing power, single board SMP modules can be wired together to form larger SMP systems. Technically, such machines are known as *cc-NUMA* designs, or cache coherent NUMA. As noted is Section A.3.1, cache coherency refers to the fact that protocols are used to ensure that local caches (i.e., on the local board) do no get out of sync with the values in a remote memory bank. For SMP models, *bus snooping* is the simplest such design [24]. Here, each processor observes read and write requests on the bus and takes appropriate action when it realizes that the current bus transaction may have some impact on the data in its local cache. In so doing, the OS/hardware can ensure that the programmer "sees" a single consistent memory image across all boards in the cluster.

Because of the simplicity of the programming model for shared memory machines, it seems very likely that the SMP model of parallel design will continue to thrive and evolve. At present, the two real drawbacks with such systems are (i) that they are still somewhat limited in scalability, and (ii) at the high end of SMP performance, the ccNUMA machines remain quite expensive.

A.5.2 The Cluster Alternative

As noted in the previous section, high performance parallel computing is still often associated with cost prohibitive hardware designs. As a result, the performance to be garnered from parallel computation is often reserved for a select group of

heavily funded organizations. In recent years, however, that environment has begun to change. Besides the introduction of commodity SMP boards, a new generation of "commodity-off-the-shelf" (COTS) distributed memory solutions has begun to emerge. In the realm of parallel computing, this has led to the deployment of systems built entirely of inexpensive PCs and network components. Known informally as *Beowulf* class clusters, the new fully distributed platforms have opened up the field of parallel computing to a wider audience than ever before [14, 15, 95, 115, 99].

In the mid-1990s, with the maturation of the commodity market (i.e., inexpensive, mass-produced products), the move to simple, low-cost cluster computing really began. These first systems were relatively low-powered, constructed mainly from x486 machines and 10Mb Ethernet; for the most part, the processors were too slow to keep up with even this simple arrangement. However, within a couple of years, the x486 machines were replaced by Pentium processors, and the network was scaled up to Fast Ethernet (100 Mb). More importantly, perhaps, the awkward "channel bonded" interconnects of the early clusters were replaced by more powerful "full wire speed" switches that have offered extremely cost-effective communication performance.

A.5.2.1 Remaining Hurdles

It should be noted, however, that the commodity market provides only a subset of the components necessary for a parallel computer, namely the CPU, memory modules, and disk storage. While these elements are important, they are not sufficient in themselves to support true high performance computing. In fact, there are several areas of active research that aim to partially, or even completely, close the gap between commodity cluster performance and that of the traditional MPP machines. The following list highlights the most important.

- Recently, bandwidth-hungry Internet applications have pushed down the price for network interconnects. In addition to the aforementioned Fast Ethernet

NAS Benchmark	Fast Ethernet	Gigabit Ethernet	Myrinet
BT	297.53	387.31	302.5
CG	81.54	127.58	154.1
MG	208.19	314.92	269.0
SP	208.87	286.07	236.2

Table A.1: NAS parallel performance in MFLOPS on 8 processors [78].

(100Mb/s), new clusters have also been built with Gigabit Ethernet (1000 Mb/s) and the even more powerful Myrinet (1.28 Gb/s). It is important to note that, unlike the broadcast-based Classical Ethernet, these new technologies can be run in *full duplex* mode. In short, this allows data to be transferred in both directions simultaneously, thereby doubling bandwidth for point-to-point connections. Table A.1 lists the MFlops performance on a number of common NAS benchmarks used for assessing parallel computation [78].

- Unfortunately, arbitrarily increasing the power of individual components in a complex system often leads to bottlenecks [118, 39]. In the current context, one of the primary bottlenecks has become the communications stack (i.e., TCP/IP in most LAN environments). Table A.2 provides a breakdown of the processing overhead associated with conventional network operations [116]. As illustrated, the dominant portion of overhead is TCP/IP processing — accounting for 48.4% of the load. [Note that the next highest total, interrupt handling (34.6%), is also affected by software]. The point to be taken here is that, no matter how fast the underlying network, the protocol stack must be able to efficiently accept, process, and transfer all data that it receives; otherwise, the gains associated with the physical layer don't translate into faster run times. Consequently, a number of *lightweight* communication frameworks have been proposed in the past ten years or so. Though each differs somewhat in logic and implementation, the common theme is the replacement of a multi-layered network kernel

Processing	Overhead	%
System call and socket	1.6 μsec	3.6
TCP	15.5 μsec	34.6
IP	6.2 μsec	13.8
Protocol handler invocation	3.2 μsec	7.1
Device driver	4.7 μsec	10.5
Hardware interrupt	5.9 μsec	13.2
NIC+Media	7.7 μsec	17.2
Total	44.8 μsec	100

Table A.2: TCP/IP Overhead [116].

with *zero-copy* protocols that transfer data directly from user space to the network interface. At present the most important standards for cluster computing include Myrinet GM [85] and the Virtual Interface Architecture (VIA) [35, 36]. Figure A.10 illustrates the performance of VIA relative to the more traditional UDP [35]. We can clearly see the advantage of using VIA over the conventional protocol at packet sizes from 32 to 1024 bytes. For cluster architectures requiring high performance — particularly when large message sizes cannot be guaranteed — communication layers like VIA and Myrinet GM offer considerable promise.

- While the previous two issues relate directly to the supporting network, another key issue involves the transfer patterns associated with disk storage. In this respect, traditional MPPs have always had a distinct advantage. A wide variety of parallel, disk-striping file systems have been developed for proprietary architectures, including PFS (Intel Paragon), PIOFS and GPFS (IBM SP), HFS (HP Exemplar), and XFS (SGI Origin). Recently, however, parallel file systems for cluster computing have begun to emerge. Perhaps the most significant of these is the Parallel Virtual File System (PVFS), a joint project of Clemson University and NASA's Goddard lab [16]. PVFS offers transparent disk striping (i.e., parallel) across optimized local file systems (i.e., virtual). Figure A.11 presents

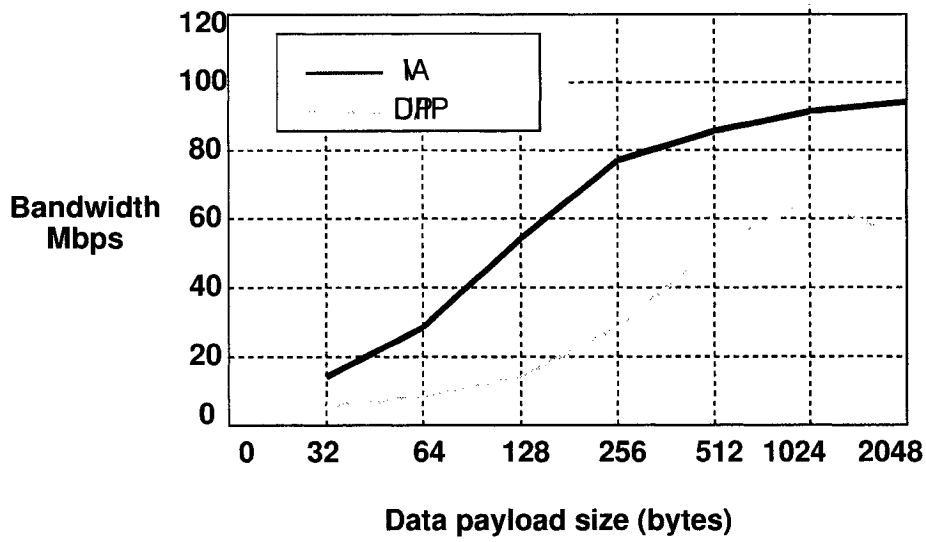


Figure A.10: Bandwidth Comparison: VI vs UDP.

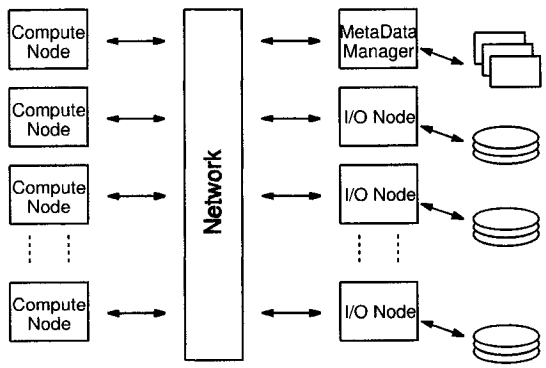


Figure A.11: The PVFS architecture.

the PVFS architecture. In short, local applications communicate directly with I/O daemons to read and write to disk files. A single metadata manager provides partitioning information for the entire parallel file system. Experimental results over both Fast Ethernet and Myrinet, reported in [16], have been very impressive.

A.6 Parallel Computing Models

A parallel computing model represents an abstraction of a class of parallel machines that attempts to capture the computational and communication capabilities of this class in terms of a small number of key parameters. In turn, these parameters represent a means by which to evaluate parallel algorithms that will be executed on the corresponding physical architectures. In the sequential setting, the ubiquity of machines adopting the basic von Neumann model has led to the wide adoption of a single, relatively simple analytic paradigm. Essentially, designers consider a platform consisting of an unlimited memory store that supports constant time access to any and all program data. Not surprisingly, this model does not work well for parallel systems; the sheer variety of such platforms precludes any single algorithmic cost model. Instead, a number of models have been suggested, each attempting to capture the salient features of a class of parallel machines [79, 91, 13, 15, 55, 72, 73, 112]. The remainder of this section discusses some of the most common parallel machine models and their application to algorithm design.

A.6.1 The PRAM

The Parallel Random Access Memory computer or PRAM [68] is an extension of the standard von Neumann model that is used so widely in the sequential setting. It is an architectural abstraction consisting of a set of p processors that access an unlimited store of global memory; any single memory word can be accessed by any processor in constant time. In addition, it has a global clock. The model can consequently be described as a synchronous (i.e., coordinated), shared memory multiprocessor. A strength of the PRAM model is that it allows the algorithm designer to focus on the challenge of task decomposition without having to worry about synchronization, communication, and the mapping of tasks to processors. There are four variants of the basic PRAM model depending upon how global memory is accessed and updated.

1. Exclusive-read, exclusive-write (EREW)
2. Concurrent-read, exclusive-write (CREW)
3. Exclusive-read, concurrent-write(ERCW)
4. Concurrent-read, concurrent-write (CRCW)

The list proceeds from most restrictive to least restrictive. In practice, it is CREW that represents the most common PRAM model for algorithm design. We further note that as *write* concurrency is added, the system becomes more complicated and "logic" is added to determine how to update memory in the face of concurrent write attempts. Typical standards are the *arbitrary* protocol (randomly choose one of the alternatives) and the prioritized protocol (a pre-defined prioritization is used).

Given that the main features of the PRAM are (i) $O(1)$ memory access and (ii) clock synchronization, it is an abstraction that most accurately models the contemporary SMP architecture. Unfortunately, even with the support of concurrency protocols, such a model is not well-suited to the distributed memory MIMD machines that are becoming more and more common. In particular, the notion that all processors have constant time access to all variables in the global memory space is clearly untrue. Use of the PRAM for distributed memory machines may lead to sub-optimal algorithm design in that the "penalty" for node-to-node communication is not adequately captured. For this reason, models representing additional classes of realistic machines have been suggested.

A.6.2 Bulk Synchronous Parallel

The BSP model was one of the first — and most significant — attempts to produce a robust cost model for practical parallel machines [121, 12]. In effect, BSP was intended as a *bridging model* between the areas of algorithm design and hardware implementation. A BSP algorithm consists of a sequence of *supersteps*, where a

superstep is defined as a local computation phase, a global communication phase, and finally a barrier synchronization. The synchronization prevents processes from proceeding beyond the barrier until *all* processes have reached this same point. This is important because it allows the local process to proceed with its computation phase, knowing that it has received all necessary data from its neighboring processes and, conversely, that they have received its messages. BSP attempts to realistically model parallel computation with three parameters.

- **l**: The time taken for a barrier synchronization. Perhaps more intuitively, it can be taken as a representation of the network latency cost since it typically represents the time for a message of minimal size to traverse the network.

- **g**: The "gap". This is the ratio of the time taken to perform a single local computational step T_L — where a step is in unit time — versus the time required to transmit a message T_M of minimal size. Intuitively, when the ratio $\frac{T_M}{T_L}$ is large, we know that we must minimize the size and quantity of message transfers within the application since communication is expensive relative to computation.

- **P**: The number of processors.

Given these three parameters, the time required for a single superstep in a BSP algorithm is formally expressed as $w + hg + l$. Here, w refers to local computation time, h to the maximum number of messages/packets sent or received during the superstep, g to the gap, and l to the time for a synchronization. The complete parallel run-time T_P for a BSP algorithm with S supersteps is simply the summation of the time required for each of the S supersteps.

BSP is notable for a couple of reasons. First, it encourages the use of *latency hiding*. In the most general sense, latency hiding refers to the capacity to mask the costs that would be associated with message transfers between concurrent tasks — the very

ones that might be identified by the PRAM model. In practice, there are a number of ways of doing this. For example, one can overlap communication with computation by sending data before it is actually required and having the processors continue to work on local tasks during the send phase. As well, one can associate groups of tasks with separate processors so that task-to-task communication occurs in an intra-node fashion (this is sometimes described by way of *virtual processes*). No matter what form latency hiding takes, the ultimate goal is the same — to exploit the fact that not all of the "atomic tasks" need be associated with concurrent communication. In parallel computing terminology, this concept is known as *parallel slackness.*

The second factor of note is that BSP makes no attempt to model the very real setup costs for message transfers. In other words, it does not explicitly discourage the use of shorter messages; they are seen as proportionally more efficient than longer ones. For this reason, we can describe BSP as a model with coarse-grained computation but (possibly) fine-grained communication.

A.6.3 LogP

A derivative of BSP, the LogP model attempts to more accurately reflect the costs of message passing architectures [23]. LogP is itself an acronym for the four parameters it supports.

- **L**: Network latency. It must be noted, however, that latency does not refer to barrier synchronization as LogP is an asynchronous model (i.e., the processors are free to proceed at their own pace).
- **o**: Message overhead. LogP tries to capture the cost of setting up a message transfer. In a sense, this is a cost that cannot be hidden; the processor must be involved in preparing the send buffer.
- **g**: The *gap* ratio (same as BSP).

- **P**: Again, the number of processors.

A number of studies have been performed on the relative value of LogP versus BSP [11]. In general, the two have been found to have roughly equivalent modelling power, though the somewhat less constrained BSP model is significantly simpler to employ in algorithm design. For this reason, LogP may ultimately be more useful as a costing model than a design paradigm.

A.6.4 CGM

Another BSP-style model is the Coarse Grained Multicomputer or CGM [33]. Like BSP, CGM builds upon the idea of computation and communication phases, followed by a barrier synchronization. However, it does not try to explicitly model the ratio of computation to communication. Instead, it assumes the existence of an arbitrary interconnection fabric that is significantly slower than local memory — making messages of any size relatively expensive. In addition, it assumes that each processor is associated with a standard local memory of size $O(\frac{n}{p})$, where $O(\frac{n}{p}) \gg O(1)$. During each superstep, a single *h-relation* of size $O(\frac{n}{p})$ can be routed. The implication in terms of algorithm design is that CGM explicitly discourages the use of many small data transfers during a given communication round. In this sense, it favours both coarse-grained computation and coarse-grained communication.

There are a number of benefits to the CGM model. First, by abstracting away the details of the network, CGM lends itself to a *portable* design model. The coarse grained communication paradigm significantly diminishes the importance of proprietary interconnects since a constant (or at most a very low degree polynomial) number of communication rounds provides such a high degree of latency hiding that the exact parameters of the network become less important.

Second, modern routing technologies have made the details of the network interconnect somewhat less significant. The older *store and forward* style of routing, in

which entire messages are collected on the switch, made hop count — and by extension network topology — a significant concern. With contemporary *cut-through* routing, however, a message is able to stream through a switch channel without collecting in a buffer, making the hop count less of a concern. As such, CGM can abstract away the network details while still providing an effective cost model.

Note that all CGM algorithms are BSP algorithms but it is not the case that all BSP algorithms are CGM algorithms. CGM is a more restrictive model that requires coarse-grained communication on those machines which are capable of efficient fine-grained communication. As such, BSP is likely to be more applicable in these cases.

A.7 Performance Measurement

Just as design methodology must be extended to account for the differences between sequential and parallel architectures, so too do the metrics for measuring algorithm performance. The two simplest — and the most commonly used measures of parallel performance — are listed below:

- **Speedup**. Speedup is simply the ratio of the sequential time T_S versus the parallel time T_P on a given number of processors p. Optimal speedup is equal to p. Theoretically, speedup can never be greater than p, though in practice this limit is sometimes violated by a phenomenon known as *super linear* speedup. Super linear speedup typically occurs when resources on the parallel machine provide the parallel algorithm with an "unfair" advantage. For example, massive memory capacity may allow the parallel algorithm to avoid the page swapping that plagues a sequential implementation.

- **Efficiency**. A second metric is the efficiency function. Expressed as $\frac{S}{P}$, the ratio of speedup to the number of processors, efficiency indicates the degree to which processors were usefully employed. A perfectly efficient algorithm has an

efficiency ratio of 1.0. For some, the efficiency metric is more "obvious" since it does not require the reader to manually factor in the processor count — the ratio "speaks for itself".

A.7.1 Non-Optimality

In practice, parallel algorithms rarely exhibit optimal speedup — 50% of optimal may be considered a success in some cases. The following list highlights the four primary factors limiting the efficiency of parallel algorithms, each of which must be understood and respected by the algorithm designer.

- **Computational Load Imbalance**. While in trivial cases perfect workload partitioning may be possible, real world algorithms are rarely this cooperative. Often, because of parameters that cannot be known or controlled at run time, certain processors may end up with a disproportionate amount of the work.

- **Inter-processor Communication**. This is pure overhead since sequential applications do no such work.

- **Redundant Computation**. Often more subtle than the first two items in the list, redundant processing typically occurs when distinct processors are not able to share the costs of specific calculations. On distributed memory machines, for example, re-calculation is often necessary since there is no way to directly access results and/or data structures on another board.

- **Inherently Sequential Computation**. Optimal speedup is only possible if 100% of the algorithm is capable of being parallelized. This is rarely true in practice (see below).

A.7.2 Scalability of Parallel Algorithms

A related performance issue is the scalability of the parallel algorithm. By scalability, we refer to the capacity of the algorithm to provide *adequate* performance as additional processing elements are added. The term "adequate" is used because desired or acceptable speedup can be an entirely subjective standard. In the most general sense we would like to see the efficiency ratio E remain relatively stable with an increase in resources. It must be understood, however, that scalability tests do not require *input size* to remain constant. On the contrary, for increases in processing resources, we allow proportional increases in input units. This is necessary due to the effect of something called **Amdahl's Law** [4]. In short Amdahl's Law says that parallel speedup cannot increase beyond the ratio of the algorithm's sequential computation to its parallel computation. In other words, speedup is fundamentally limited by the proportion of the code that is inherently sequential. So if, for example, 20% of the original algorithm exhibits no exploitable concurrency, then the maximum speedup is $1/0.2 = 5$, regardless of the number of processors available.

A.8 Application Support

Traditionally, vendors of parallel hardware systems have also provided compilers, development tools, and communications libraries with their products. Optimized for the underlying hardware, they represented an efficient but non-portable means by which to develop parallel applications. For this reason, attempts to standardize programming models were undertaken. Perhaps the most significant achievement in this respect is the Message Passing Interface (MPI) [83]. MPI is an industry-backed standard that defines a communications API for distributed memory architectures. It is not a software product itself, but instead defines the components, functions, and parameters that must be present in an MPI implementation. Vendors are free to implement the library in system-specific ways, but code compiled for one platform

is guaranteed to run on any other platform that conforms strictly to the standard. This uniformity and portability has been a tremendous boon to developers of parallel programs.

A.8.1 MPI Primitives

As noted, MPI is not a language. Instead, it is a rich collection of functions combined into a library that may be linked to user code. At its core, MPI exploits an intuitive send/receive communication model. Processes manage their own local memory spaces and communicate explicitly with other remote processes when data sharing is required. For maximum flexibility, transfers may be either *blocking* (the call returns when the appropriate buffer is ready for re-use) or *non-blocking* (control is returned immediately to the calling program which must later check to see if the call has completed).

The original MPI-1 standard, published in 1994, extends this simple concept with support for collective communication operations (i.e., many nodes participating in a single logical exchange). Such operations are fundamental to MIMD programs and include:

- **broadcast**: Distribute a message from one process to all cooperating processes.
- **reduce**: Combine values from p processes into a single value on the local node using a binary operator. Global sum is an example.
- **gather**: Collect values from p processes into a local array of size p.
- **scatter**: Distribute the local values of an array of size p to p cooperating processes.

In response to user requests, the standard was extended again in 1997, with MPI-2 providing additional functionality with respect to such things as *dynamic process creation*, *parallel I/O*, and *one-sided remote memory access*.

With its broad support amongst users, researchers, and vendors alike, MPI has become the *de facto* standard for distributed memory parallel computing. It should be noted, however, that while MPI is very well-suited to distributed memory systems, it has also been implemented on most high-end SMP or ccNUMA machines as well. On such systems, explicit TCP/IP message passing does not take place. Instead, portions of the global address space are reserved as local *transmission buffers* and message transfers are executed efficiently as memory-to-memory copies.

A.8.2 MPI Alternatives

In addition to MPI and the vendor-specific offerings, a number of other programming models are currently being employed. A precursor of MPI, the Parallel Virtual Machine (PVM) [98] , is a communication subsystem that originated in the academic community. Despite offering functionality that is roughly equivalent to MPI, PVM is becoming somewhat less popular due to the fact that it does not support a recognized standard.

A.8.3 SMP Support

For shared memory architectures, standardization efforts have led to the OpenMP specification [89], begun in 1997. Like MPI, OpenMP has the support of a broad collection of industrial partners and is rapidly gaining acceptance within the multiprocessor community. Unlike MPI, with its rich collection of callable library functions, OpenMP uses compiler-based directives in combination with a run-time library to exploit parallel resources. Specifically, a *fork-join* model is used. An OpenMP program executes as a single program until a programmer-designated parallel region is encountered. The code "forks" into multiple threads which then execute independently across processors, each using some portion of the global address space. When the threads complete their work, they are terminated and the original master program

continues execution until either it finishes itself or another group of threads is required. As a simple example of such a model, programmers may thread the iterations of a loop if the computation within each iteration is independent of that of other iterations.

In using shared memory, OpenMP can, in theory at least, provide a decidedly less demanding programming paradigm. However, it should be understood that OpenMP and MPI are associated with fundamentally different architectural foundations. Just as MPI may be *overkill* on a shared memory model due to the coding complexity of explicit memory accesses, OpenMP is not applicable in a shared-nothing environment. As such, we can expect both models to continue to flourish in the near term.

A.9 Conclusion

This chapter has examined the historical foundations of parallel computation, as well as the issues particularly relevant to the research described in this thesis. Unlike the sequential programming paradigm, which has enjoyed an enviable uniformity of design and implementation patterns, its parallel counterpart exhibits much greater variability. Algorithm designers must not only be familiar with the basic principles of asymptotic analysis, but they must also understand the performance impact of such things as memory architectures, network design, and communication characteristics. In short, parallel software implementation can be a more complex and challenging process. Having said that, the benefits can also be significant. Not only do parallel implementations allow researchers to compute solutions more quickly but they also permit new computational research that would likely not have been undertaken in their absence. The *data cube* research discussed in this thesis is one such example. By exploiting parallel computing models, both from an algorithmic and implementation-oriented perspective, we are able to more effectively address the enormous computational and I/O requirements often associated with current data

warehousing problems.

Appendix B

The Theory of NP-Completeness

In Appendix A, Section A.7, we noted that load imbalance is one of the primary sources of non-optimality in a parallel algorithm or application. In many cases, finding a good load balancing can be reduced to finding a good partitioning of some form of task graph into p equal-sized pieces. Unfortunately, many such partitioning problems are known to be NP-complete. Because our own partitioning/scheduling strategies rely upon task graphs, and because NP-completeness has important implications for algorithm design, this appendix provides a brief introduction to the theory.

To analyze an algorithm is to formalize the performance characteristics of the algorithm in terms of key parameters. Most often, we *bound* the run-time of an algorithm as a function of its input size n. If that function is polynomial in n — $O(n^k)$ where k is a constant — we can assume that the associated algorithm is *tractable* on real systems. (We note that while $O(n^{50})$ is polynomial but likely intractable, virtually all common polynomial time algorithms are of low degree). Polynomial time bounds therefore provide us with a convenient yardstick by which to compare the performance of alternative algorithmic designs.

While the majority of computing problems that one is likely to encounter in practice are amenable to such polynomial time algorithms, there is in fact a sizable group of important problems that have resisted all attempts to design tractable algorithmic solutions. We note that while "algorithms" for these problems do exist, these

algorithms typically consist of some form of "brute force" or exhaustive search of the solution space and consequently run in exponential or factorial time. More importantly, however, not only have no polynomial time solutions been found, but no one has been able to determine whether tractable algorithms are even possible. In other words, it is not yet certain whether this set of problems is intrinsically distinct from the polynomially solvable problems.

Despite this, a formal model has been developed that allows researchers to say something meaningful about the relative "hardness" of computational problems [46, 61, 22]. Specifically, the model considers what are known as *decision problems*, those problems whose output can be represented in the form of a "yes/no" answer. (We note that while this requirement may seem somewhat artificial, it poses no practical problem since most common algorithms can be recast in this manner.) Within this context, we may define the *complexity class* P as consisting of the set of languages $L \subseteq \{0,1\}^*$ such that there exists an algorithm A that decides L in polynomial time. Because a full treatment of language theory is beyond the scope of this thesis, we simply note the following about the preceding definition. First, a language L consists of a set of strings defined on the *binary* alphabet $\{0,1\}^*$. Second, an algorithm *decides* a language L if for any such binary string, it either accepts or rejects that string as belonging to L. Finally, since a computational problem is ultimately associated with a concrete encoding — typically represented as a binary string — we may for convenience use the terms "language" and "problem" interchangeably. The complexity class P, therefore, represents exactly those decision problems whose solutions can be provided by polynomial time algorithms.

As noted, however, there exists a large collection of problems for which polynomial time solutions are not known. To deal with this issue, the complexity model is extended by introducing the concepts of *certificates* and *verification algorithms*. Effectively, a certificate is a string that can be used to *prove* that a particular decision

problem has an affirmative answer. In turn, a verification algorithm is one that, given a problem instance *and* a certificate, provides such a proof in polynomial time. As a very simple concrete example, consider the decision problem, "Does a directed graph G have a path whose length is exactly four?" Clearly, given the graph G and a purported example of a four-edge path as input, the verification algorithm can determine in polynomial time whether the original question has an affirmative answer.

We now define the *complexity class NP* as consisting of those problems (formally, languages) for which a polynomial time verification algorithm exists. Simply put, a decision problem belongs to the class NP if it is possible to determine in polynomial time whether a potential solution actually solves the problem. (We should note that the term NP is an acronym for non-deterministically polynomial. Formal language theory describes such an algorithm as one that is capable of guessing possible solutions and then confirming them in polynomial time. An indefinite number of guesses can be carried out concurrently/non-deterministically; it is important only that the confirmation procedure be polynomially bounded. Conceptually, however, this definition is equivalent to the one used in this thesis). It should be clear that all problems belonging to the complexity class P fit this definition since we can obtain a valid certificate for these problems simply by executing their existing polynomial time algorithms. Formally, then, $P \subseteq NP$. The open question, of course, is whether $P = NP$?

While the answer to this question is not currently known, there is compelling support for the belief that P and NP are distinct. This support comes by virtue of the NP-complete problems/languages. A problem Q is considered NP-complete if (i) $Q \in NP$ and (ii) every other problem in NP is *polynomial time reducible* to Q. We note that a problem $Q\prime$ is reducible to a problem Q if there exists an algorithm that transforms the encoding of $Q\prime$ into an encoding for Q. A solution to Q is therefore a solution to $Q\prime$ within a polynomial factor (in terms of time). Given

this definition, the second criterion for NP-completeness would seem to be virtually impossible to satisfy. In 1971, however, Stephen Cook showed that *any* problem in NP could be reduced in polynomial time to *one* specific problem in NP, namely the logic-oriented *Satisfiability* problem [21]. As such, Satisfiability became the *first* NP-complete problem. The existence of one NP-complete problem provided a much simpler mechanism for proving NP-completeness for other problems. Specifically, a problem Q can be proven NP-complete simply by demonstrating a reduction from any other NP-complete problem $Q\prime$ (originally the Satisfiability problem) since $Q\prime$, by definition, is reducible from any other problem in NP.

The true significance of NP-completeness theory, however, is that it implies that a polynomial time solution for *any* problem in NP would imply a solution for *all* problems in NP. This is so, of course, since a problem in NP *without* a polynomial time solution could simply be reduced to a problem *with* a polynomial time solution. However, since no polynomial time solution has ever been found for an NP-complete problem, it appears extremely unlikely that such problems can be solved in a tractable amount of time. By extension, we regard the NP complete problems as the "hardest" problems in NP, since solving any of them in polynomial time is at least as hard as solving any other problem in NP.

This understanding has important implications for researchers and algorithm designers. It suggests that if a new problem is provably NP-complete, then attempting to find an optimal solution is almost certain to be unsuccessful. Techniques for dealing with such problems in practical settings might include:

1. If the problem size is very small, an exhaustive search of the solution space may be feasible.

2. Identify a given problem as a *special case* of a general NP-complete problem, where the special case is amenable to a polynomial time solution.

3. Develop an approximation algorithm that attempts to find a solution that, while not provably optimal, is both efficient and provides acceptably good answers.

In practice, then, by using one of these approaches, it is often possible to provide good solutions to problems that would appear to be intractable. In fact, in this thesis we see how both Item 2 and Item 3 in the preceding list are used to tackle partitioning and scheduling problems.

Appendix C

Multi-dimensional Indexing Techniques

Chapter 5 presented the details of a fully distributed, high performance indexing framework for the OLAP setting. The RCUBE was, in fact, based upon a multi-dimensional indexing structure known as the R-tree. In this appendix, we review some of the most important approaches to multi-dimensional indexing, with a particular emphasis on those techniques designed for disk-based indexing in relational environments.

C.1 The Origin of Indexing

Almost since their inception, one of the most important applications of computer systems has been for the storage and processing of vast amounts of collected data. To support such processing, it quickly became apparent that linear searches of the accumulated data records would be a terribly inefficient means by which to query the data. What was needed, then, was an indexing scheme that could be used to quickly locate selected records.

As it turned out, however, indexing was not a challenge to be taken lightly. In addition to the daunting size of even the early databases, one of the key problems was generating indexes that could respond to the dynamic nature of practical systems.

In fact, it was not until 1963 that Adel'son-Vel'skii and Landis developed a self-adjusting tree that could be used to retrieve leaf/data nodes in logarithmic time [2]. By guaranteeing that the height of the left and right children in the modified binary tree could differ by at most one, the *AVL tree* became an effective indexing mechanism for main memory.

Nonetheless, it took another decade for researchers to find disk-based methods that could work appropriately for much larger datasets. In 1972, Bayer and McCreight proposed the *b-tree*, a broad shallow search tree — similar in flavour to a binary search tree — that could be optimized for disk-block retrievals [5]. During the same time period, various hashing techniques (e.g., linear hashing, extendible hashing) were also proposed and found their way into commercial implementations.

The story of multi-attribute indexing mechanisms, on the other hand, continues to unfold. As it turns out, the single dimension methods just described do not extend terribly well, if at all, to higher dimensions. Instead, what is needed are new mechanisms that can support orthogonal range queries (among other types of queries) that concurrently identify the boundaries of that query in each of the d dimensions. An example of a simple multi-dimensional query is given in Figure C.1.

This geometric interpretation of the data space has proven to be extremely difficult to model. A variety of techniques, most based in some way on single dimension or in-memory models, have been developed and used with moderate success [45]. In general, we can say that each method enjoys some success in a particular environment. Beyond that we can also say with certainty that no existing technique has been consistently successful in high dimension spaces. This so-called "curse of dimensionality" has resisted the efforts of the best researchers for more than twenty years.

Nevertheless, despite the limitations of the current methods for orthogonal range searching, there are still plenty of environments in which they are indispensable. Such areas include GIS systems, data warehousing, multi-media databases, and even

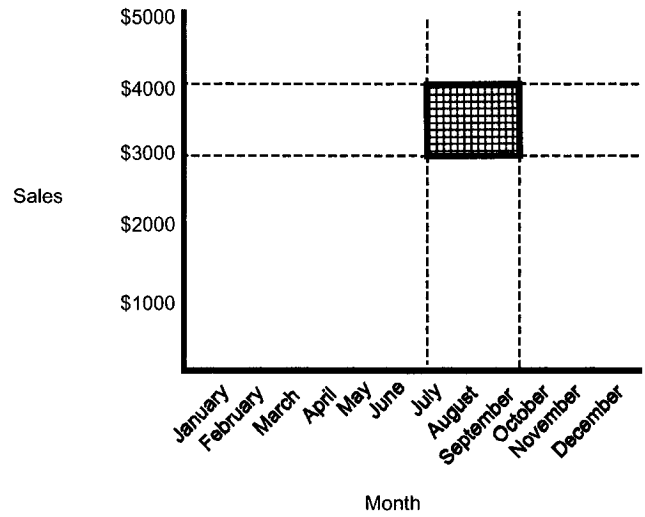


Figure C.1: A simple multi-dimensional query. In this case the query identifies those sales made during the third quarter whose individual value was between 3000 and 4000 dollars.

Web indexing. In the remainder of this section, we look at a number of the more common, and successful, techniques. We note that the focus will be on the conceptual underpinnings of the access methods rather than a review of pseudocode for particular algorithms. We look initially at the original indexing methods designed primarily for main memory. The more important (at least in the current context) disk-based methods will then be discussed, with a particular emphasis on grid-based, hierarchical, and space-filling curve methods.

C.2 In-core methods

As mentioned above, the AVL-tree was one of the first techniques that could be used to efficiently manage data housed within a computer's core memory. Since then a number of other indexes have been developed, each meeting with a certain measure of success. In this section, we will look at the main features of three of the most familiar of these mechanisms.

The **quadtree** [104, 41] is one of the more common designs and can be described as

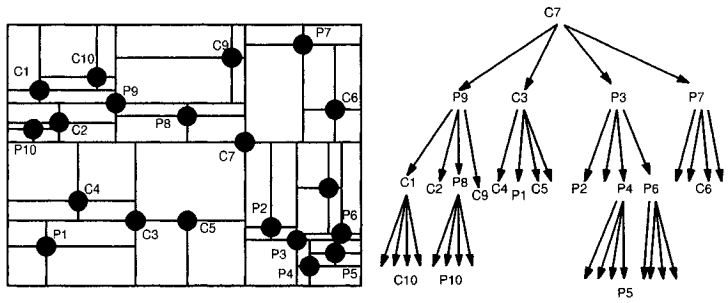


Figure C.2: The point quad-tree.

a multi-dimensional generalization of the simple binary (i.e., one parent-two children) search tree. In this case, however, the interior nodes have 2^d descendants, each corresponding to an interval shaped partition of the given subspace. Figure C.2 provides a graphical illustration. In a quadtree each node has seven fields (four child pointers, X and Y coordinates, and a key name). In k dimensions — the three dimensional version is known as an *oct-tree* — the node size becomes $k + 2k + 1$. As with the binary tree, points are used as partitioning values as well as data values.

Insertion algorithms use X and Y values to find the proper partition. The new data point is then used to divide the vacant region into four sub-regions, generally of unequal size. In the average case, the cost of insertion (and the simpler search) are equivalent to the *total path length* (TPL) which is $O(\log_4 n)$. In the worst case, each point is inserted into the deepest node and we have $O(n)$. For the search, worst case time is $O(n^{1-1/k} + d)$ where d is the number of points reported.

In some cases, the impact of the worst case can be reduced. If all data is known beforehand, a multi-feature sort can ensure that the tree doesn't reduce to a "linked list" type of architecture. It is also possible to dynamically re-order the tree by partially rebuilding it when some pre-defined balance criterion is violated. For larger spaces, point quadtrees require testing of all k keys at each accessed node. Also, the node sizes become quite large.

While insertion is relatively straightforward, deletion can be quite expensive —

even for two dimensions — since the data points are used to partition the space. If such points are removed, major re-adjustments are required to restructure the subtree below that node.

The **k-d-tree** [7] is another variation on the binary search tree in which the "test" attribute at successive levels varies from d_i to d_k (see Figure C.3 for an illustration). Each node in a 2^d k-d-tree contains six fields (two pointers to the left and right son, the X and Y coordinates, a key name, and the name of the appropriate test attribute). In k dimensions, we have $4 + k$ fields, where k is the number of coordinates.

For the most part, insertion is similar to that required in binary search trees. We split the parent region on X, then split the child on Y, etc. Average cost of inserting is $O(\log_k n)$. Again, the worst case results in $O(n)$ time. Like the quad-tree, *a priori* knowledge of the data set can minimize the impact of worst case insertion (this requires sorting). In terms of searching, the worst case can be given as $O(n^{1-1/k} + d)$ where d is the number of points reported (i.e., the same as the quadtree).

A second *static* approach is the "adaptive k-d tree". Here, data is stored only at the leaf nodes and the interior nodes contain the median of the set along a given dimension. The discriminator key is the one that provides the greatest "spread" across all values. (The same key may be used at successive levels). We recursively break the set down until only a small number of values remain — these are then placed into a linked list. In general, the tree is *more* balanced but, still, imbalances can and do result when several nodes have a *tie* on some value.

Deletion in adaptive k-d trees is quite complicated since we may need to repartition the data. In fact, deletion is quite involved for any type of k-d tree.

Generally speaking, the k-d-tree is faster than the point quadtree since only one key comparison is done at each level. As well, it has more efficient storage — only two pointers per node instead of 2^k.

A somewhat more complex extension of the binary search tree is the **range tree**

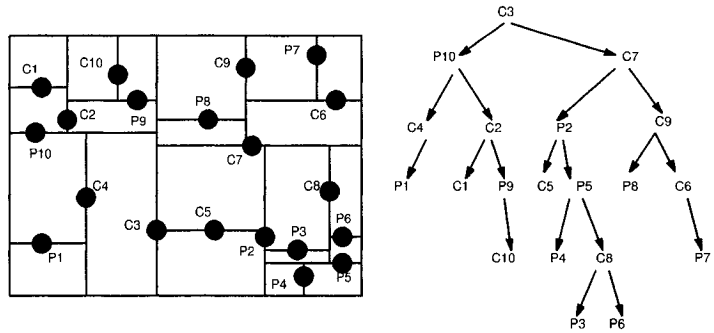


Figure C.3: The k-d-tree.

[8]. We construct the data structure by first sorting along one of the dimensions and then building a balanced binary search tree in which the leaf nodes are connected as a doubly linked list. Figure C.4 provides an illustration. For each of the interior nodes, we build another binary search tree in which we sort all children of the interior node along the next dimension. We continue recursively in this fashion until we have moved through all k dimensions.

To perform a range search, we select the ancestor node that is closest to the node delimiting the "low" range value and the"high" range value (i.e., the most recent common ancestor). For children in the left sub-tree, we select those nodes whose left child is also greater than the low value. For each of these nodes we recursively search on the right sub-tree. For all children in the right sub-tree of the common ancestor, we apply the same process (in reverse, of course). In k dimensions, range search takes $O((\log_2 n)^k + F)$, where F is the number of points found.

The performance comes at a price, of course. Namely, the total storage for all these binary search trees is $O(n \log_2 n)^{k-1})$. In short, the range tree is fast but very large in k dimensions.

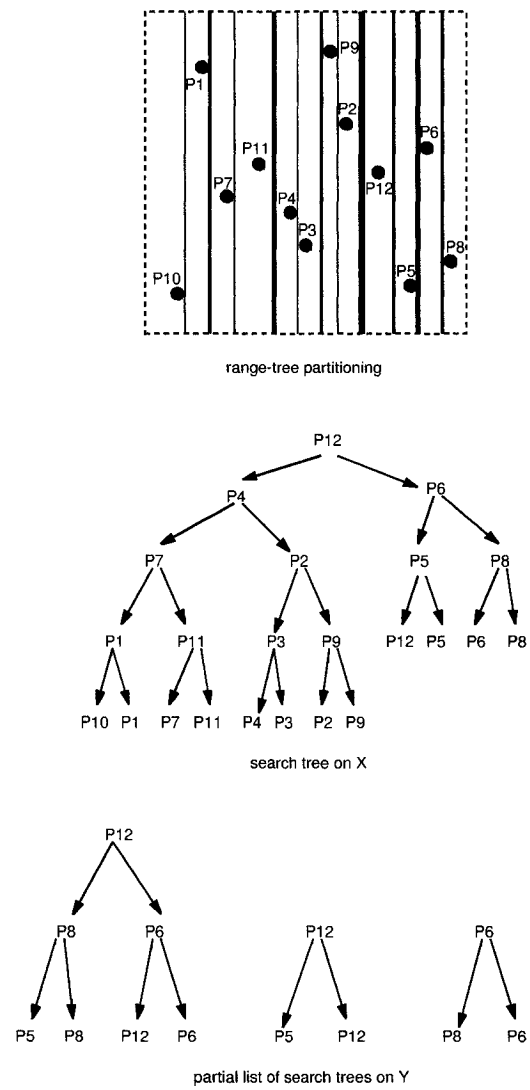


Figure C.4: The range tree. The upper figure shows the space partitioned by x-value. The thickness of the vertical lines denotes the level of the binary tree.

C.3 Disk-based Methods

It might seem rather logical to extend the above methods into environments in which the data sets are large enough to require permanent disk storage. Unfortunately, this does not work due to the access characteristics of mechanical hard disk units. In particular, bits are retrieved from disk in a block-by-block fashion in order to reduce the *average* cost of satisfying a disk request. In other words, when a byte or group of bytes is required, the operating system reads an entire disk block into memory. This makes sense, of course, when we consider the fact that the most expensive component of a disk access is the time taken to position the read/write head; once in place, the transmission proceeds quite quickly. Furthermore, since it is likely that the data/bytes surrounding the original request will also be required, significant performance gains can be had by dumping a contiguous chunk of data into main memory at one time.

With this in mind, it is clear that indexing methods that operate in disk-based environments must be constructed so as to exploit this feature. For example, a binary tree would give terrible performance if ported directly to disk-oriented environments in that a branching factor of two would introduce a large number of disk head movements when traversing the tree.

Loosely speaking, we can place the relevant techniques into three general classes — those based upon grids, those based upon trees, and those based upon space filling curves. In this section, we will look at examples of each.

C.3.1 Multi-dimensional Hashing

Perhaps the most natural or intuitive manner in which to divide a space is to partition it into some form of grid. A query can then be answered by "hashing" the request to produce a pointer to a *data bucket* that contains (or may contain) the actual record. The data bucket in this case corresponds to a disk block.

The **grid file** [87] super-imposes a k-dimensional irregular grid over the data set.

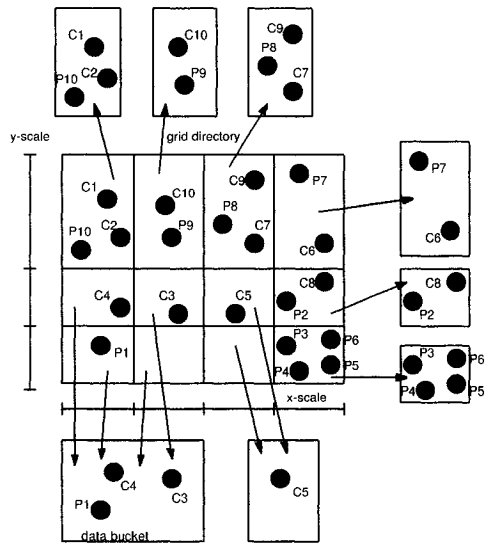


Figure C.5: The grid file.

Each *cell* in the set is associated with a block-sized *bucket* on disk; the bucket, in turn, may contain more than one cell. A *directory* (i.e., a k-dimensional array) locates the cell within a given bucket. Since there are more cells than buckets, this grid array can grow relatively large and is typically stored on disk. See Figure C.5 for an illustration of the basic technique.

For performance reasons, we would like to keep some portion of the directory in main memory. Consequently, the directory actually consists of two components: (a) the aforementioned array and (b) k one-dimensional arrays called *scales*. The scales break the grid array into sections corresponding to the width of each cell along a given dimension.

When a point search is specified, we look at the in-memory scales to determine the coordinates of the relevant cell in the grid array. A single disk access is necessary to retrieve the proper segment of the array. Using the grid array, we then obtain a pointer to the disk bucket and retrieve the block with a second disk access. In general, the grid file guarantees a maximum of two disk accesses for any given point.

For range queries, we must retrieve all blocks that overlap the search area. Since

the range may be of arbitrary size and the cells are not of fixed width, there is no meaningful boundary on the number of disk blocks that must be retrieved.

When a bucket overflows, it needs to be split. With a grid file, this means that a new hyper-plane (of size $k-1$) will have to be inserted into the grid array in order to reduce the size of the bucket. However, this also means that all cells/buckets along this plane will be divided as well, resulting in the "global" modification of the grid array (the same is true for deletion). In the worst case, there may be $O(n^k)$ grid entries. This property can introduce a super-linear size increase in the directory when individual points are added to the data set. In the case of a range search, more blocks may be accessed to read the directory than to read the final data blocks.

The grid file's average directory size for uniformly distributed data is $O(n^{1+(k-1)/(kb+1)})$ where k is the bucket size. The average space occupancy of the buckets is 69% — as indicated by the original authors (note: testing was only conducted in two and three dimensions).

The grid-file is the earliest and perhaps most well known of the hashing techniques. Various modifications have been developed, however. These include the *EXCELL* method [119] which decomposes the point universe into a regular grid (as opposed to the arbitrary hyperplanes of the conventional grid file), the *two-level grid file* [59] that uses a second grid file to manage the grid directory, and the *twin grid file* [63] which uses a pair of balanced grid files to span the entire space (unlike the two-level grid file that arranges the two files hierarchically). While each of these methods may offer marginal improvements in given environments, their merits have not been significant enough to elevate the grid-file to the position of a *de facto* indexing standard.

C.3.2 Hierarchical Tree-based Methods

Because the hashing methods often lead to very large directory structures, a more appealing option for higher dimensions may be the exploitation of hierarchical indexing

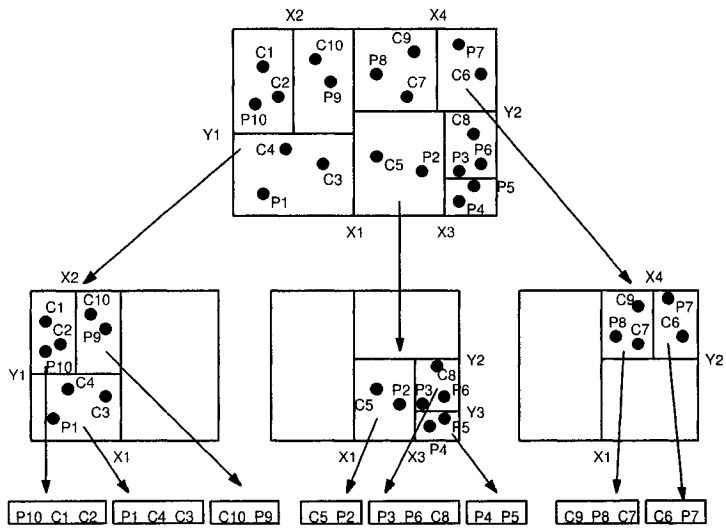


Figure C.6: The k-d-b-tree.

techniques. Here, our goal is to produce a tree-like structure that can support logarithmic access time, even on secondary memory devices. Generally speaking, each of the methods discussed below represents some form of binary tree or b-tree extension to higher dimensional space.

A **k-d-b-tree** [100] partitions the space in the style of the adaptive k-d-tree (i.e., it uses median values to recursively partition the space into two equal sub-spaces). Figure C.6 provides an illustration. Each interior node represents an interval shaped region and data points are stored in the leaf nodes.

Like b-trees, the tree grows from the leaves upward. Points are added until a region exceeds the upper limit. When it does, the area is split along the dimension that offers the broadest distribution of points (i.e., we try to split the node so that the area, as well as the number of points, is equalized). As a result, it is height balanced.

In practice, the k-d-b tree does not grow as cleanly as one would like. There are two reasons for this. First, though it is assumed that a space can always be split more or less evenly along a given dimension, this is not always the case (e.g., think of a set of points that form a "plus" shape). The resulting split creates unbalanced sub-trees.

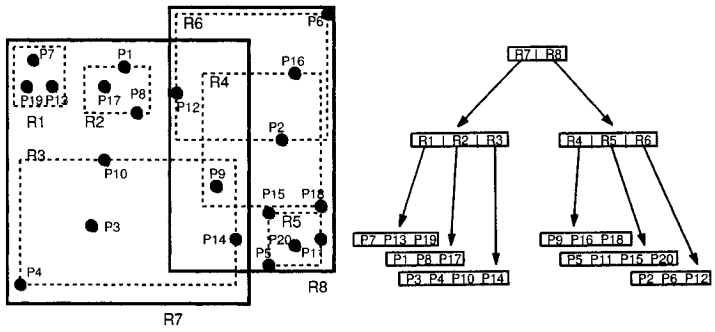


Figure C.7: The R-tree.

Second, and more importantly, the k-d-b-tree exhibits what are called *cascading splits*. When a group of child nodes exceeds some maximum limit, the parent node must be split in half. If this happens, however, the new split will almost certainly cut through the boundaries of its children. The end result is that splits will cascade down through the tree, carving children into non-optimal blocks. Since the blocks may be severely under-sized, the k-d-b-tree can guarantee no minimum space requirements (except in the special case of a one-dimensional b-tree).

The **R-tree** [53] is another multi-dimensional implementation of the b-tree. In this case, nodes in the tree represent maximum bounding boxes (MBB) that recursively include either smaller MBBs or the data points themselves. Figure C.7 provides a graphical illustration.

MBBs within a given parent may represent non-contiguous regions. Likewise, they are free to overlap or intersect one another. A MBB contains M references to child MBBs. When a node exceeds this limit it is split into two new nodes, each of size m, where $m = M/2$. On the other hand, should deletions reduce the child count to less than m, the node is split and the child boxes are re-distributed to nearby MBBs.

When existing boxes must be expanded in order to encapsulate a new point, the algorithm is constructed so as to minimize the area covered by the new configuration. Three techniques are presented in the original paper: an exponential, quadratic, and linear mechanism. In general, the quadratic approach is the one taken for practical

implementations.

As such, we can say that the maximum fan-out of a node is M, while the minimum or worst-case fan-out is $m = M/2$ (note: the denominator in this expression can be varied but 2 is a common setting).

The height of the tree can now be easily established. In the best case, it is $O(\log_M n)$. Conversely, in the worst case, it will be $O(\log_{M/2} n)$.

The R-tree is attractive because of its clean worst-case storage bounds and the simplicity of its insertion, deletion, and splitting routines. However, it suffers from the fact that searches often require an excessive number of path traversals due to the fact that MBBs can overlap. This is particularly expensive when doing range searches. The original author downplayed this fact by providing experimental results that showed very good search performance characteristics. It must be noted, however, that the results were produced on just two dimensions and with very small data sets (less than 5000 rectangles).

C.3.3 Space-filling Curves

In order to index high dimensional space, both grid-files and tree-based methods rely upon data structures and algorithms that work directly within the d-dimensional universe. An alternative approach is to *compress* the larger space down into a single dimension and then use a standard index like a *b-tree* [22] to answer user queries. Conceptually, a space filling curve is like a thread that winds its way through a d-dimensional spatial grid such that (i) the thread intersects every point in the grid and (ii) every grid point is visited exactly once. The order of traversal represents a linear ordering on the point space that can then be indexed by way of the b-tree.

Ideally, a good space filling curve should locate points in this one dimensional space such that immediate neighbours would have been neighbours in the d-dimensional

space as well. While there is no total order that completely preserves spatial proximity, a primary objective of any space filling curve is to minimize the size of "jumps" between any two consecutive points since long jumps ultimately place un-related points into the same disk block. The following list describes four common space fillings curves in terms of their distinguishing features (see Figure C.8 for a graphical illustration of the techniques):

- Row-wise scan: A very simple mechanism that is likely to be inappropriate for anything but raster images, as the jumps between grid points can be extremely long.

- Peano or Z-ordering: An interleaving of the bits of the binary representation of the X and Y coordinates in two-dimensional space (this, of course, can be extended to higher dimensions).

- Gray-coding: Here, the grid points are coded into binary representations such that successive/contiguous locations on the curve differ in exactly one bit position.

- The Hilbert curve: Formally, the Hilbert curve is drawn by way of a string rewriting grammar [93] that identifies a recursive sequence of forward/backward, left/right movements, along with angle transitions. Informally, it is a very tight curve that produces jumps whose length never exceeds three.

For point queries, space filling curves provide very good response time since a d-dimensional point can be directly mapped to a scalar value — the *index* of the space filling curve — which can then be retrieved in logarithmic time from the b-tree structure. For range queries, however, the mapping is much more complex. Specifically, because the curve may wind in and out of the query rectangle, it is necessary to identify the blocks containing points that may lay inside the query space.

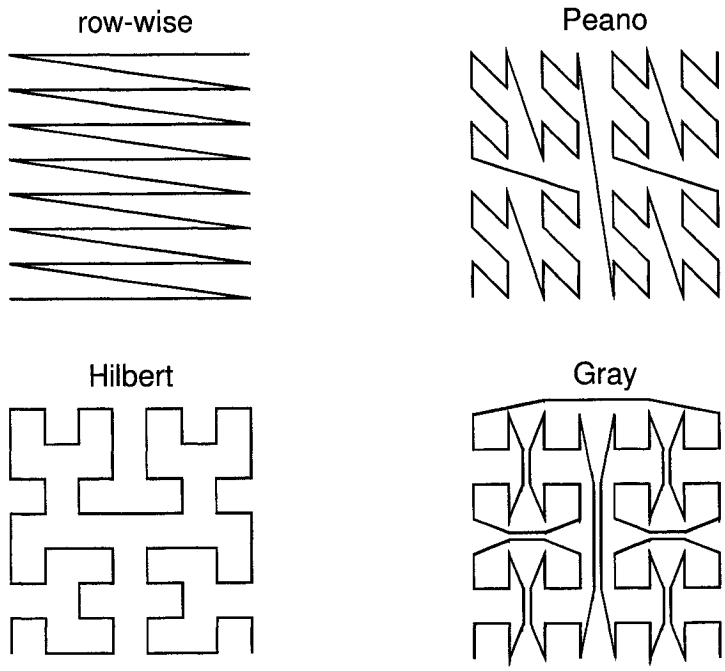


Figure C.8: Common space filling curves.

This requires a large number of expensive "spaced-filling curve to scalar" mapping operations that can seriously affect performance, particularly in high dimensions.

A further problem in high dimensions is that the size of the "key" values may exceed the capacity of standard machine words. In particular, since the range of the scalar index values is equivalent to the cardinality product $\prod_{i=1}^{d} C_i$, the index may have to maintain and process arbitrary length integers, again negatively affecting performance.

C.4 Comparative Results

While the number of multi-dimensional indexing techniques described in the literature probably exceeds 100 [45], most are not widely used. Only a few, for example, have found their way into commercial products. The most obvious of these include the R-tree (Informix [64]) and the z-order space filling curve (Oracle [90]).

In terms of comparative evaluation, a number of researchers have undertaken

"head-to-head" studies. Unfortunately, definitive conclusions are difficult due to the absence of a standard benchmark, as well as variations in implementation quality. Nevertheless, a number of general observations can be made. The R-tree (and its variants) are by far the most heavily studied of all multi-dimensional access methods. Furthermore, the R-tree often serves as the benchmark by which more experimental methods are evaluated, suggesting that the R-tree has received the widest level of acceptance in the research community. In terms of specific evaluations, Smith and Gao [113] demonstrate that the R-tree provides significantly better query performance than the grid file and z-order based indexes, while Greene [51] shows that the R-tree is vastly superior to the k-d-b-tree. With respect to the space filling curves, most research seems to indicate that the z-order and Hilbert curves are most appropriate for the purposes of database indexing [67, 1].

Appendix D

The Data Generator

In order to support a robust experimental environment, we have written our own data generation sub-system. In fact, the data generator has evolved into a significant application in its own right. Because of the number of optional parameters that need to be passed to the generator application, coupled with the desire to maintain a permanent record of specific data set "schemas", we have constructed the data generator with the compiler tools FLEX and BISON, the GNU equivalent of LEXX (lexical analysis) and YACC (statement parsing). In so doing, we allow users to define a data set schema — in the form of an ASCII text file — that identifies data set parameters as *key/value* pairs. This file is parsed by the generator front-end to produce a final specification set that, in turn, is passed to the back-end component that actually produces the data set. In other words, we have defined a small data set specification language (DSSL) that provides tremendous flexibility in terms of defining data characteristics. More importantly perhaps, the language-based mechanism can be extended in the future to provide even more sophisticated specification options.

Figure D.1 lists the current BISON DSSL specification. As noted, the job of the BISON system is to parse the contents of the input file by breaking it into a sequence of valid DSSL statements. Formally, the specification is what is known as a *grammar*. BISON, in fact, accepts languages that can be represented by a particular type of grammar, namely the *context free* grammars. BISON-compatible languages

```
START:
   /* empty */
   |  line
;
line:
   spec
   |  line spec
;
spec:
   W IN_FILE {strcpy(out_file, $2)};
   | R NUM {row = $2};
   | D NUM {dimension = $2 + 1};
   | S NUM {seed = $2};
   | Z NUM {zipf_alpha = (double)$2};
   | Z FLOAT {
       strcpy(zipf_string, $2};
       atof(zipf_string);
     }
   | COLUMN DIM {cardinality[$1 - 1] = $2};
;
```

Figure D.1: The current BISON DSSL specification.

are therefore known as context free languages. (Note: formally, BISON actually accepts a subset of context free languages known as *LALR(1)*, but a full treatment of this distinction is far beyond the scope of this thesis). Very briefly, a context free language can be represented as a set of rules or *productions*, which in turn are composed of a finite set $(V \cup T)$ of *variables* V and *terminals* T. A terminal can be defined as the simplest element of a language — a native word. By contrast, variables are defined recursively in terms of terminals and other variables. In a context free language, a production defines the relationships between the variables and terminals and has the form $A \Rightarrow \beta$ (read A *derives* β). Here, A represents a *variable*, while β defines a *string* of variables and terminals. A BISON-generated parser *accepts* a language L by recursively decomposing or *reducing* an input set (i.e., a string of terminals) to a special *START* symbol.

As a simple but concrete example, let's assume that our user-defined schema contains the string "r 10000". Since the BISON DSSL grammar contains a production of the form $spec \Rightarrow RNUM$, then the string "r 10000" can be reduced to the variable symbol *spec* when R is substituted for "r" and NUM is substituted for "10000". A

sequence of such transformations reduces the input set to a collection of *line* variables, which in turn are finally reduced to the START symbol. We note that, in order to perform these production reductions, the BISON parser relies upon a supporting mechanism that *tokenizes* the input into its constituent terminal symbols (e.g., R and NUM). In the current case, this lexical analysis facility is provided by FLEX, a GNU tool that uses a regular expression paradigm to define the format of all terminals. Finally, we add that once the input file has been accepted and all data set parameters have been captured, the code fragments associated with each production in the BISON grammar are then responsible for replacing the default parameter values with those defined in the schema file.

A typical data cube schema file is presented in Figure D.2. In this example, we define a data set with 100,000 rows, 8 dimensions (plus an implicit 9^{th} dimension for the measure attribute), an output (or write) file called "sample.dat", a mildly skewed zipf distribution, and a range of cardinalities defined on each of the nine fields in each record. The skew component itself utilizes the *zipf* power-law function [123]. Here, we express the probability of encountering a particular value i in a given dimension d as $P_i \sim 1/i^a$, where $1 \leq i \leq C_d$ and the probability P is normalized into the range $0 \ldots 1$. Recall that C_d is the cardinality of d. As the *zipf* factor a is incremented from zero, the data set becomes more skewed. Typical values of a are 0.0 to 1.0.

As noted, we can easily extend the grammar to include more sophisticated parameter specifications. For example, the line "c01 [c 14 z 0.2]" could indicate an attribute with a uniquely defined cardinality *and* a uniquely defined skew pattern. In summary, the current data set generation facilities are both rich and flexible, and permit thorough evaluation of all of the algorithms we currently study.

```
r 100000 d 8
w sample.dat
z 0.1
c01 10
c02 4
c03 50
c04 12
c05 12
c06 2
c07 25
c08 8
c09 100
```

Figure D.2: Example of data set schema.

Appendix E

An Algorithm for Distributed Index Generation

In this appendix, we describe a parallel algorithm for building the distributed indexes described in Chapter 5, Section 5.4. Here, we assume that each individual view starts out localized on a single processor, that is, as part of the output of the generation algorithms in Chapters 3 and 4. The details are presented in Algorithm 30.

Much of the complexity of the algorithm is actually associated with the striping phase. While we could simply stripe each of the views of the data cube in sequence (i.e., one at a time), such an approach would generate a series of $p * j$ *point to point* communication rounds, where j is the total number of views to be indexed. (For the full data cube, $j = 2^d$). In such a model, $p - 1$ links would go unused during each stripe transfer as a single partition was being sent from the source node to exactly one of the destination nodes. We improve network utilization by overlapping the striping traffic across each of the p processors. To do so, we order the views to be striped in decreasing order by size. Each node informs its neighbors of the number of views on its local disk; the maximum local view count represents the total number of striping rounds. During each of these rounds, each node partitions its current view and, in p communication *phases*, sends its p stripes to the appropriate destination processors. In a synchronized fashion, it receives a distinct stripe from each of its $p - 1$ neighbors, and uses them to create p partial R-tree indexes (the local processor

Algorithm 30 Distributed Index Generation

Input: A subset of views S on each of p nodes, generated by the existing Parallel Data Cube Generation algorithm. Collectively, the p subsets form a set B.

Output: Distributed R-tree indexes for each of the views of B.

```
1: On each node, create a list L of the views in S.
2: Sort L in descending order by view size.
3: listSize ⇐ number of views in L
4: Collect listSize from neighboring p − 1 processors
5: maxCount ⇐ maximum listSize of neighbors
6: for i = 1 to maxCount do
7:   if i < maxCount then
8:     viewToSend ⇐ yes
9:     Sort the view in Hilbert order
10:  end if
11:  source ⇐ dest ⇐ myRank
12:  for each processor p do
13:    if viewToSend then
14:      Distribute stripe_p to dest
15:    end if
16:    if listSize of source < maxCount then
17:      Collect data stripe into buffer
18:      Construct R-tree Index from buffer
19:    end if
20:    source incremented clockwise
21:    dest decremented counter-clockwise
22:  end for
23: end for
```

creates a partial index from a portion of its current view as well). To avoid deadlock, the communication transfer pattern is ordered as a pair of concurrent loops — source processors in a clockwise sequence and destination processors in a counter-clockwise sequence.

We note that most good MPI implementations should provide an *all to all* collective operation that is structured in this manner. However, these library functions cannot be used in this context because the data cube generation algorithm described in this thesis only seeks to balance the total "workload" assigned to each processor — it does not necessarily guarantee an identical number of views on each node. Typically, there will be a difference of one or two small views from node to node. As a result, a *hand rolled* collective striping operation must be written so as to allow specific nodes to *opt in or out* of the final few striping rounds, depending upon whether (i) the local node has any views remaining to be partitioned and (ii) the designated source node has any views left to send to the local node. The *viewToSend* flag and the *(listSize of source* $<$ *maxCount)* conditional comparison are used for this purpose.

We note that the algorithm described here has a non-optimal number of rounds. Specifically, the number of communication rounds is a function of p. Recall that a fundamental goal of the CGM model of parallel algorithm design is to produce algorithms with a fixed number of communication rounds. In so doing, the overhead for message setup and delivery of the first byte (i.e., latency) is minimized. In fact, the striping algorithm can be written so as to consist of a number of communication rounds independent of p. For example, it is possible to first sort all views on each node, then package the sorted results into a single list to be distributed to each processor. However, because the size of the sorted list of views is likely to be far larger than main memory, the list would have to be incrementally stored to disk and read back in to memory prior to transfer. The I/O cost would be significant and would almost certainly exceed the setup costs for the additional messages required of the current

algorithm.

Bibliography

[1] D. Abel and D. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographic Information Systems*, 4(1):21–31, 1990.

[2] G. Adelson-Velskii and E. Landis. An algorithm for the organization of information. *Soviet Math Doklady 3*, 3:1259–1263, 1962.

[3] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. *Proceedings of the 22nd International VLDB Conference*, pages 506–521, 1996.

[4] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conference Proceedings*, pages 483–485, 1967.

[5] R. Bayer and M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[6] R. Becker, S. Schach, and Y. Perl. A shifting algorithm for min-max tree partitioning. *Journal of the ACM*, 29:58–67, 1982.

[7] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[8] J. Bentley. Decomposable searching problems. *Information processing letters*, 8:244–251, 1979.

[9] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest-neighbour meaningful? *Lecture Notes in Computer Science*, 1540:217–235, 1999.

[10] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. *Proceedings of the 1999 ACM SIGMOD Conference*, pages 359–370, 1999.

[11] G. Bilardi, K. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs LogP. *ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32, 1996.

[12] R. Bisseling and W. McColl. Scientific computing on bulk synchronous parallel architectures. *IFIP Congress*, pages 509–514, 1994.

[13] G. Blelloch and B. Maggs. Parallel algorithms. *ACM Computing Surveys*, 28(1):51–54, 1996.

[14] R. Buyya, editor. *High Performance Cluster Computing: Architectures and Systems*, volume 1. Prentice, 1999.

[15] R. Buyya, editor. *High Performance Cluster Computng: Analysis of Algorithms*, volume 2. Prentice Hall, 1999.

[16] P. Carns, W. Ligon, R. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[17] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26:65–74, 1997.

[18] E. Codd. Is your DBMS really relational? and Does your DBMS run by the rules? *ComputerWorld*, 1985.

[19] E. Codd, S. Codd, and C. Salley. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Technical report, E.F. Codd and Associates, 1992.

[20] Comshare. http://www.comshare.com/.

[21] S. Cook. The complexity of theorem proving procedures. *ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[22] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, 1996.

[23] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. Eicken. LogP: towards a realistic model of parallel computation. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[24] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1998.

[25] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the datacube. *International Conference on Database Theory*, 2001.

[26] F. Dehne, T. Eavis, and A. Rau-Chaplin. A cluster architecture for parallel data warehousing. *International Conference on Cluster Computing and the Grid (CCGRID 2001)*, 2001.

[27] F. Dehne, T. Eavis, and A. Rau-Chaplin. Coarse grained parallel on-line analytical processing (OLAP) for data mining. *Proceedings of the 2001 International Conference on Computational Science (ICCS 2001)*, pages 589–598, 2001.

[28] F. Dehne, T. Eavis, and A. Rau-Chaplin. Computing partial data cubes for parallel data warehousing applications. *Euro PVM/MPI 2001*, 2001.

[29] F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallelizing the datacube. *Distributed and Parallel Databases Journal: Special Issue on Parallel and Distributed Data Mining*, 2001.

[30] F. Dehne, T. Eavis, and A. Rau-Chaplin. Top-down computation of partial ROLAP data cubes. Submitted for publication, 2003.

[31] F. Dehne, T. Eavis, and A. Rau-Chaplin. Top-down computation of partial ROLAP data cubes. Journal Version: Submitted for publication, 2003.

[32] F. Dehne, Todd Eavis, and A. Rau-Chaplin. Distributed multi-dimensional ROLAP indexing for the data cube. *The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003.

[33] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *ACM Symposium on Computational Geometry*, pages 298–307, 1993.

[34] B. Dinter, C. Sapia, G. Hofling, and M. Blaschka. The OLAP market: State of the art and research issues. *ACM First International Workshop on Data Warehousing and OLAP*, pages 22–27, 1998.

[35] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, 1998.

[36] T. Eicken and W. Vogels. Evolution of the Virtual Interface Architecture. *Computer*, 31(11):61–88, 1998.

[37] Oracle Express. http://otn.oracle.com/products/express/content.html.

[38] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. *Symposium on Principles of Database Systems*, pages 247–252, 1989.

[39] P. Farrell and H. Ong. Communication performance over a gigabit ethernet network. *The IEEE International Performance, Computing, and Communications Conference*, pages 181–189, 2000.

[40] W. Feller. *An Introduction to Probability Theory and its Applications.* John Wiley and Sons, 1957.

[41] R. Finkel and J. Bentley. Quadtrees: A data structure for retrieval of composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[42] P Flajolet and G. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[43] M Flynn. Very high speed computing systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[44] G.N. Frederickson. Optimal algorithms for tree partitioning. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 168–177, 1991.

[45] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[46] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[47] S. Goil and A. Choudhary. High performance OLAP and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, (4), 1997.

[48] S. Goil and A. Choudhary. High performance multidimensional analysis of large datasets. *Proceedings of the First ACM International Workshop on Data Warehousing and OLAP*, pages 34–39, 1998.

[49] S. Goil and A. Choudhary. A parallel scalable infrastructure for OLAP and data mining. *International Database Engineering and Application Symposium*, pages 178–186, 1999.

[50] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Proceeding of the 12th International Conference On Data Engineering*, pages 152–159, 1996.

[51] D. Greene. An implementation and performance analysis of spatial data access methods. *In Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 606–615, 1989.

[52] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for OLAP. *Proceeding of the 13th International Conference on Data Engineering*, pages 208–219, 1997.

[53] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD Conference*, pages 47–57, 1984.

[54] X. Huang H. Lu and Z. Li. Computing data cubes using massively parallel processors. *7th Parallel Computing Workshop (PCW '97)*, 1997.

[55] S. Hambrusch. Models for parallel computation. *Proceedings of Workshop on Challenges for Parallel Processing: International Conference on Parallel Processing*, 1996.

[56] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, 2000.

[57] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes. *Proceedings of the 1996 ACM SIGMOD Conference*, pages 205–216, 1996.

[58] P. Hass, J. Naughton, S. Seshadri, and L. Stokes. Sampling based estimation of the number of distinct values of an attribute. *Proceedings of International VLDB Conference*, pages 311–322, 1995.

[59] K. Hinrichs. Implementation of the grid file: Design concepts and experience. *BIT*, pages 569–592, 1985.

[60] Hitachi, 2003. http://www.top500.org/ORSC/1996/node10.html.

[61] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[62] HPCVL, 2003. http://www.hpcvl.org.

[63] A. Hutflesz, H. Six, and P. Widmayer. Twin grid files: Space optimizing access schemes. *ACM SIGMOD International Conference on Management of Data*, pages 183–190, 1988.

[64] INFORMIX INC. The DataBlade architecture, 1997. http://www.informix.com.

[65] W. Inmon. *Building the Data Warehouse.* John Wiley, 1992.

[66] K. Iverson. *A Programming Language.* John Wiley, 1962.

[67] H. Jagadish. Linear clustering of objects with multiple attributes. *ACM SIGMOD International Conference on Management of Data*, pages 332–342, 1990.

[68] J. Jaja. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[69] I. Kamel and C. Faloutsos. Parallel r-trees. *Proceedings of ACM SIGMOD*, pages 195–204, 1992.

[70] I. Kamel and C. Faloutsos. On packing r-trees. *Proceedings of the Second International Conference on Information and Knowledge Management*, pages 490–499, 1993.

[71] N. Koudas, C. Faloutsos, and I. Kamel. Declustering spatial databases on a multi-computer architecture. In *Proceedings of Extended Database Technologies*, pages 592–614, 1996.

[72] V. Kumar, A. Grama, A. Gupta, and G. Karypsis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[73] V. Kumar and A. Gupta. Analysis of scalability of parallel algorithms and architectures: A survey. *International Conference on Supercomputing*, pages 396–405, 1991.

[74] LEDA, 2003. http://www.mpi-sb.mpg.de/LEDA/.

[75] R. Lehn, V. Lambert, and M. Nachouki. Data warehousing tool's architecture: From multidimensional analysis to data mining. *Workshop on Database and Expert Systems Applications*, pages 636–643, 1997.

[76] T. Leighton. Methods for message routing in parallel machines. *ACM Symposium on Theory of Computing*, pages 77–96, 1992.

[77] S. Leutenegger, M. Lopez, and J. Eddington. STR: a simple and efficient algorithm for r-tree packing. *Proceedings of the 14th International Conference on Data Engineering*, pages 497–506, 1997.

[78] J. Mache. An assessment of gigabit ethernet as a cluster interconnect. *IEEE International Workshop on Cluster Computing*, pages 36–42, 1999.

[79] B. Maggs, L. Matheson, and R. Tarjan. Models of parallel computation: A survey and synthesis. *Proc. of the 28th Hawaii International Conference on System Sciences (HICSS)*, 2:61–70, 1995.

[80] K. Mehlhorn and S. Naher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[81] MP1, 2003. http://www.top500.org/ORSC/1996/node15.html.

[82] MP2, 2003. http://www.top500.org/ORSC/1996/node16.html.

[83] The Message Passing Interface standard, 2003. http://www-unix.mcs.anl.gov/mpi/.

[84] S. Muto and M. Kitsuregawa. A dynamic load balancing strategy for parallel datacube computation. *ACM 2nd Annual Workshop on Data Warehousing and OLAP*, pages 67–72, 1999.

[85] Myrinet GM, 2003. http://www.myri.com.

[86] R. Ng, A. Wagner, and Y. Yin. Iceberg-cube computation with PC clusters. *Proceedings of 2001 ACM SIGMOD Conference on Management of Data*, pages 25–36, 2001.

[87] J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[88] The OLAP Report. http://www.olapreport.com.

[89] OpenMP, 2003. http://www.openmp.org.

[90] ORACLE INC. Advances in relational database technology for spatial data management: A White Paper, 1995.

[91] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *ACM Symposium on Theory of Computing*, pages 510–513, 1998.

[92] Paragon, 2003. http://www.top500.org/ORSC/1996/node35.html.

[93] H. Peitgen and D. Saupe. String rewriting systems. *The Science of Fractal Images*, pages 273–275, 1988.

[94] Y. Perl and U. Vishkin. Efficient implementation of a shifting algorithm. *Discrete Applied Mathematics*, (12):71–80, 1985.

[95] G. Pfister. *In search of clusters.* Prentice Hall, 1998.

[96] Pilot. http://www.marketwave.com/products_solutions/pilot_suite/pas.html.

[97] Programming POSIX threads. http://www.humanfactor.com/pthreads.

[98] The Parallel Virtual Machine, 2003. http://www.epm.ornl.gov/pvm/.

[99] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the power of parallelism in a Pile-of-PCs. *Proceedings of the IEEE Aerospace Conference*, 1997.

[100] J. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. *ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[101] K. Ross and D. Srivastava. Fast computation of sparse data cubes. *Proceedings of the 23rd VLDB Conference*, pages 116–125, 1997.

[102] N. Roussopolis and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. *Proceedings of the 1985 ACM SIGMOD Conference*, pages 17–31, 1985.

[103] N. Roussopoulos, Y. Kotidis, and M. Roussopolis. Cubetree: Organization of the bulk incremental updates on the data cube. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 89–99, 1997.

[104] H. Samet. The quadtree and related hierarchical data structure. *ACM Computing Surveys*, 16(2):187–260, 1984.

[105] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, California, 1996.

[106] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. *International Conference on Data Engineering*, pages 328–336, 1994.

[107] B. Schnitzer and S. Leutenegger. Master-client r-trees: A new parallel architecture. *11th International Conference of Scientific and Statistical Database Management*, pages 68–77, 1999.

[108] SGIPC, 2003. http://www.top500.org/ORSC/1996/node23.html.

[109] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1990.

[110] A. Shukla, P. Deshpande, J. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. *Proceedings of the 22nd VLDB Conference*, pages 522–531, 1996.

[111] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: shrinking the PetaCube. *Proceedings of the 2002 ACM SIGMOD Conference*, pages 464–475, 2002.

[112] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[113] T. Smith and P. Gao. Experimental performance evaluations on spatial access methods. *In Proceedings of the Fourth International Symposium on Spatial Data Handling*, pages 991–1002, 1990.

[114] SP2, 2003. http://www.top500.org/ORSC/1996/node34.html.

[115] T. Sterling, J. Salmon, D. Becker, and D. Saverese. *How to build a Beowulf.* The MIT Press, 1999.

[116] S. Sumimoto, H. Tezuka, A. Hori, T. Takahashi, and Y. Ishikawa. High performance communication using a commodity network for cluster systems. *The Ninth International Symposium on High-Performance Distributed Computing*, pages 139–146, 2000.

[117] T3E, 2003. http://www.top500.org/ORSC/1996/node29.html.

[118] T. Tam and C. Wang. Contention-free complete exchange algorithms on clusters. *IEEE International Conference on Cluster Computing*, pages 57–64, 2000.

[119] M. Tamminen. The extendible cell method for closest point problems. *BIT*, 22:27–41, 1982.

[120] TOP500, 2002. http://www.top500.org.

[121] L. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.

[122] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multi-dimensional aggregates. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 159–170, 1997.

[123] W. Zipf. *The Psycho-Biology of Language: An Introduction to Dynamic Philology.* Houghton Mifflin, 1935.