# OPTIMIZATION OF MULTIPLE-DISJUNCT QUERIES IN A RELATIONAL DATABASE SYSTEM

by
M. Muralikrishna

Computer Sciences Technical Report #750
February 1988

# OPTIMIZATION OF MULTIPLE-DISJUNCT QUERIES IN A RELATIONAL DATABASE SYSTEM

by

M. Muralikrishna

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1988

This research was partially supported by the Department of Energy under contract DE-AC02-81ER10920, by the Defense Advanced Research Projects Agency under contract N00039-86-C-0578, by the National Science Foundation under grants DCR-8512862, MCS82-01870, and MCS81-05904, and by a Digital Equipment Corporation External Research Grant.

## ABSTRACT

In this thesis, we describe the optimization of arbitrarily complex queries expressed in relational calculus. The qualification list is allowed to be any complex boolean expression involving both ANDs and ORs. In other words, the qualification list may have an arbitrary number of disjuncts. The query graph of each disjunct may also have any number of components. Optimizing the various disjuncts independently of each other can be very inefficient. Considerable savings in cost can be achieved by optimizing the various disjuncts together.

In a multiple-relation multiple-disjunct query, it may be possible to combine two or more disjuncts into one term. This will cut down the number of scans on each relation and also the number of times each join is performed. The objective will be to merge the disjuncts into the minimum number of terms. Minimizing the number of terms can be formulated as the problem of covering a merge graph with the minimum number of complete merge graphs, which are a restricted class of cartesian product graphs. The problem of minimizing the number of terms is NP-complete. We present polynomial time algorithms for special classes of merge graphs. We provide a heuristic for general merge graphs.

For single-relation multiple-disjunct queries involving more than one attribute, an optimal access path might consist of more than one index. The cost in our optimization model, for single relation queries, is measured in terms of the number of pages fetched from disk. We will formulate the problem of finding a set of optimal access paths for a single-relation multiple-disjunct query as one of finding a minimum weighted vertex cover in a hypergraph. Finding the cheapest vertex cover in a hypergraph is NP-complete. We present a new approximation algorithm that gives near optimal vertex covers for random hypergraphs over a wide range of edge probabilities.

We also demonstrate the usefulness of equi-depth multi-dimensional histograms in optimizing queries using multi-dimensional indices.

## ACKNOWLEDGEMENTS

I have been extremely fortunate to have had the opportunity to work with Prof. David DeWitt. Prof. DeWitt has always provided a very stimulating research environment for all his students. Whenever I needed it, he has willingly provided me with technical and emotional support. Without his very deep conviction in my abilities to do research, this thesis would not have been possible. I would like to thank him for making my stay in Madison a very fruitful and memorable one.

I also wish to thank all the members of my committee: Mike Carey, Yannis Ioannidis, Anne Condon, and John Beetem. Their suggestions and comments have gone a long way in improving the quality of this thesis.

I had the good fortune of taking a course in advanced databases under Prof. Carey. His encouragement was instrumental in my pursuing doctoral work in the database area.

I am grateful to Prof. Ioannidis and Prof. Eric Bach for suggesting innumerable enhancements, especially in those chapters that are of a theoretical nature. I would like to thank Prof. Bach for having taught me a formal course in the theory of computer science.

I am deeply indebted to Prof. Udi Manber for suggesting the hypergraph paradigm for finding the optimal access path for single-relation multiple-disjunct queries and for the proof of NP-completeness in Section 4.9.

I would like to thank my friends Ashok and Giri for their technical and moral support throughout the period of my graduate study in Madison. It was truly wonderful to discuss a wide range of topics with Ashok and also to hear him sing. Giri has always found the time to listen to my 'outrageous' ideas. I have been fascinated by Giri's ability to come up with counter examples for my *theorems*! I have also benefited from discussions with Darrah and Kirk.

Outside of school, a number of people have helped in making me feel at home in Madison. My friends and roommates (through the years) Jeff, Jenny, Jim, Mark, Rich, Steve, Tom, and Wai-Sze deserve special mention. Their affection, understanding, and willingness to put up with me for so many years has been amazing. I will cherish their friendship for a long time to come. I am extremely grateful to my host

families, the Balus and the Raos, who have so warmly welcomed me to their homes on many an occasion.

I would be amiss, if I did not acknowledge how fortunate I am to have a family that has provided me with the strength and love to pursue my goals. But for their inspiration, commitment, and sacrifice, graduate study would not have been possible. The values and morals that my family has instilled in me through the years have been invaluable to me. Amma, Nana, Buchi, and Kitta: thank you so very much. I love you all.

TABLE OF CONTENTS

# CHAPTER 1

# INTRODUCTION

## 1.1. Motivation

Query optimization[1] in relational databases has been an active area of research for well over a decade. However, most of the previous work has dealt with the optimization of single disjunct (conjunctive) queries with a single component [Smith75, Wong76, Selinger79, Yao79, Youssefi79]. In these queries the qualification list consists of selection clauses and join clauses separated only by AND connectors. Furthermore, the query graph has only a single component. [Selinger79] provides a comprehensive technique for the optimization of single-disjunct, single-component queries with an arbitrary number of relations. Many industrial strength optimizers, such as System R, RTI Ingres, IDM etc., have been built using the techniques of [Selinger79]. Since the techniques for optimizing conjunctive queries are well known, it seemed natural to evaluate a multiple-disjunct query by optimizing each disjunct separately [Bernstein81, Kerschberg82]. The result of the entire query could then be obtained by taking the union of the results of each disjunct. Optimizing each disjunct separately in a multiple-relation query or a single-relation query can, however, be very inefficient.

In this thesis we present a systematic technique for the optimization of multiple-disjunct, multiple-component queries. Our cost, for single relation queries, is measured in terms of the number of pages fetched from disk. Not considering CPU cost for single-relation queries is justified since the IO cost is generally proportional to the CPU cost.

We must point out that we do not perform any semantic or global optimization. Also, in discussing the optimization of multiple-relation queries, we do not worry about the exact methods by which the joins are performed. Instead, we show how we can reduce the number of times each join is performed by combining disjuncts.

---

[1] As is customary in most of the literature on Query Optimization, the goal is not necessarily to find the optimal access plan but to obtain an access plan that is as close to the optimal as possible.

## 1.2. An Overview of the Thesis

In Chapter 2, we will briefly survey the previous work in query optimization in relational databases. Our survey will reveal that there has been little work done in optimizing queries involving multiple-disjuncts.

The optimization of multiple-relation, multiple-disjunct queries is presented in Chapter 3. We will show how we may combine or merge two or more disjuncts into one term. Merging disjuncts reduces the number of scans for each of the relations present in the query and also the number of times each join has to performed. We will show that the problem of minimizing the number of terms is NP-complete. For a certain restricted class of queries, we will present polynomial time algorithms for minimizing the number of terms. We will present a heuristic for the general case.

We will then discuss, in Chapter 4, the optimization of single-relation multiple-disjunct queries. Efficient optimization methods can be developed for relations with no more than two relevant indices. Access planning for a query with three or more indices is NP-complete. We will show that deriving an optimal access plan for a single-relation, multiple-disjunct query can be formulated as finding a minimum cost vertex cover in hypergraphs.

In Chapter 5, we present a new approximation algorithm for finding a minimum cost vertex cover in simple graphs and extend the algorithm to hypergraphs. We will show that our approximation algorithm performs very well for random graphs and hypergraphs over a wide range of densities or edge probabilities.

In Chapter 6, we demonstrate the usefulness of using equi-depth multi-dimensional histograms when optimizing queries in the presence of multi-dimensional indices. We will present two schemes for estimating the number of tuples within a query box. The usefulness of the random sampling technique in building equi-depth histograms will be discussed.

Finally, we present a summary of our results and a list of topics for future research in Chapter 7.

## 1.3. Terminology and Notation

In this section we introduce some basic definitions.

A **query** Q(TL, QL) in QUEL [Stonebraker76] consists of a target list TL and a qualification list QL. The TL is a list of (range_variable[2].attribute_name) pairs separated by commas. The QL is a boolean expression tree whose leaves are either select clauses or join clauses and whose internal nodes are either ANDs or ORs. We assume that NOTs are removed from the QL by a preprocessor through repeated applications of DeMorgan's Law[3].

A **join clause** is a triplet of the form {r.a, θ, s.b} where a and b are attributes of r and s respectively (r and s are range variables). This clause represents the join condition r.a θ s.b, where θ is a relational operator such as '=','≠', '≤' etc.

The **query graph** for some single-disjunct query Q is denoted by QG(Q). QG(Q) is an undirected (with no self loops) graph G = (V, E) where the vertices V are the range variables referenced in QL or TL. E = {r--s | {r, θ, s} is some join clause in QL that references both r and s}. QG may be a multi-graph[4] if there is more than one equi-join clause between the same pair of range variables.

QG may or may not be connected. Each connected part of the QG is called a **component**. The result of the query is the cartesian product of the results of all the components. By definition, each range variable in Q, each clause in QL, and each (range_variable.attribute_name) in TL belongs to a unique component.

A component Φ is an **if-component** if the set Rel($TL_\Phi$) is empty. Rel ($TL_\Phi$) denotes the set of range variables present in TL that correspond to component Φ.

The **disjunctive normal form** of the boolean expression corresponding to QL is denoted by DNF(QL). It consists of one or more disjuncts separated by ORs. Each disjunct consists of clauses separated by ANDs.

A **single relation expression** is any boolean expression where the selection clauses are all on the same relation. Note that a single relation expression may have multiple disjuncts.

---

[2]If every relation in the query has a unique range variable, then relation can be used in place of range variable.

[3]NOTs can be removed from clauses by using complementary relational operators. For example NOT(sailor.age > 20) is equivalent to sailor.age ≤ 20.

[4]A multi-graph is a graph in which there may be more than one edge between the same pair of vertices.

The **range** [lower-bound, upper-bound] of an attribute in a disjunct is defined as the range of values (lower-bound ≤ attribute ≤ upper-bound) that the attribute can take in the result tuples of the disjunct.

The **selectivity factor** of a boolean expression $\alpha$ on relation R, denoted by SF ($\alpha$, R) or SF($\alpha$), is the fraction of tuples in R that satisfy $\alpha$. We will use SF($\alpha$) whenever R is clear from context.

A **bipartite graph** G is an undirected graph whose vertex set can be partitioned into two disjoint subsets X and Y, such that each edge has one end in X and the other end in Y. A bipartite graph is commonly denoted by (X, E, Y) where E is the set of edges in the bipartite graph.

A **complete bipartite graph** (X, E, Y) is one in which each vertex of X is joined to each vertex in Y.

A (loopless) graph is said to be **k-colorable** if there is an assignment of k colors, 1, 2, ..., k, to the vertices of the graph such that no adjacent vertices have the same color.

A **chordless four cycle** is a subgraph with four vertices $v_1$, $v_2$, $v_3$, and $v_4$ such that the only edges present in the subgraph are $v_1$--$v_2$, $v_2$--$v_3$, $v_3$--$v_4$, and $v_4$--$v_1$.

A graph G on n vertices is said to belong to the $G_{n, p}$ class of random graphs if each of the $\frac{n(n-1)}{2}$ edges is included in G independently and with probability p, $0 < p < 1$.

A **clique** in a graph is a set S of vertices such that each pair of vertices in S is connected by an edge.

An **independent set** in a graph is a set S of vertices such that no pair of vertices in S is connected by an edge.

A **vertex cover** in a graph G = (V, E) is a subset S of V such that every edge in E is incident with at least one vertex in S.

The **adjacency matrix** A for a graph with n vertices is a n x n matrix, where each A[i, j] is defined as follows:

$$A[i, j] = \begin{cases} 0 & \text{if the ith vertex is not adjacent to the jth vertex.} \\ 1 & \text{if the ith vertex is adjacent to the jth vertex.} \end{cases}$$

A square, integer matrix is called **unimodular** if its determinant is equal to 1 or -1. An integer matrix A is called **totally unimodular** if every square, nonsingular submatrix of A is unimodular.

Other definitions will be introduced as required.

# CHAPTER 2

## LITERATURE SURVEY

The purpose of a DataBase Management System (DBMS) is to shield a computer user from the details of secondary storage management. One major criticism of early DBMSs was their inefficiency in carrying out complex operations. A role of the Query Optimizer is to help in alleviating this inadequacy. Relational DBMSs have been very successful in shielding the user from worrying about how the data is laid out at the storage level. However, this shifts the responsibility of finding an optimal way of accessing the data from the user to the DBMS, thus making the DBMS more complex. In this chapter, we will briefly review the previous work on query optimization in relational[5] systems. An excellent starting point for this review is presented in [Jarke84].

It has been shown that optimization of even simple conjunctive queries is NP-complete [Chandra77, Aho79, Yao79]. Current systems take a top-down approach to optimization and treat it in a uniform and heuristic manner [Wong76, Selinger79, Youseffi79].

User queries can be represented in various standard forms. **Relational calculus** [Codd72] is a notation for defining the result of a query through the description of its properties. The QUEL language [Stonebraker76] is based on relational calculus. **Relational algebra** is a collection of operators on relations. Some of these operators are the traditional set operators such as union, intersection, difference, and cartesian product, while others are special relational algebra operators, such as, restriction, projection, join, and division. A sequence of relational algebra operators defines an algorithmic sequence for the construction of the result of a query. Relational calculus, on the other hand, is a non-procedural representation of the result of a query. It has been shown in [Codd72] that any relational calculus expression can be converted into an equivalent relational algebra expression. **Query graphs** have also been used to represent queries. Two classes of graphs have been used: **object graphs** [Youseffi79, Bernstein81a] and **operator graphs** [Smith75, Yao79]. Nodes in object graphs represent relation variables and constants. Edges describe

[5]See [Codd70] for more on the relational model.

predicates that these objects must satisfy. Operator graphs describe an operator-controlled data flow by representing operators as nodes that are connected by edges indicating the direction of data movement. **Tableaus** [Aho79] have been used to represent a subset of relational calculus queries. These queries involve only equality based selections, projections, natural-joins, and only AND connectors. Thus, tableau queries are a subset of conjunctive queries [Chandra77]. The number of rows in the tableau (excluding the summary row) is one more than the number of joins in a query. Semantic information (functional dependencies) is used to reduce the number of rows and hence the number of joins in a query. Tableaus have been of theoretical interest only as they are not general enough to represent all possible relational expressions. For example, they are strictly less powerful than Query By Example (QBE) [Zloof77]. We will not discuss tableaus in any more detail.

Most query optimizers start by transforming the user query into some standard canonical form. This standard form is then rearranged/simplified into an equivalent but cheaper expression. Two standard forms that have been used are the disjunctive normal form (DNF) and the conjunctive normal form (CNF). DNF has been used in [Bernstein81, Kerschberg82] to optimize and evaluate the query disjuncts separately. Both Ingres [Wong76] and System R [Selinger79] seem to use CNF but do not explain how they deal with queries that have ORs in their qualification list. All of their examples (as in other query optimization papers) deal only with queries that have no ORs. Single disjunct queries can be augmented with additional selects using transitive rules [Youseffi79]. As pointed out earlier, since the general optimization problem is computationally intractable, cheaper expressions are obtained by applying heuristics. A series of projections can be combined into a single projection and similarly a sequence of restrictions can be combined into a single restriction [Smith75]. Minimizing the sizes of intermediate results is achieved by performing selections and projections before joins and cartesian products [Smith75, Wong76]. Experimental results [Youseffi79] have shown that performing selections before joins is a very good heuristic. Significant improvements can sometimes be achieved by using semantic information (such as integrity constraints) [Hammer80, King81]. However, no industrial strength query optimizers that make use of semantic information have been built so far.

[Klug82] extends the power of relational calculus by formally incorporating **aggregate functions.** [Kim82] discusses the optimization of **nested queries.** [Dayal87] presents a unified approach to process-

ing queries that contain nested subqueries, aggregates, and quantifiers.

Optimization of single-relation, single-disjunct queries is a well explored area. One-dimensional index structures such as the B-tree [Bayer72, Comer79, ISAM, and hash indices [Fagin79, Litwin80] are commonly used for accessing only the relevant part of a relation. [Schkolnik85] shows that there is a tradeoff between the number of indices and the cost of updating those indices and that most optimizers ignore the cost of updating indices during access planning. Multi-dimensional structures such as grid files [Nievergelt84], R-trees [Guttman84], and KDB-trees [Robinson81] have also been developed for accessing only the relevant subset of the database. Applying the selection clauses to tuples of a relation can be optimized by producing threaded code [Gries71].

Queries involving two relations (one join) have been extensively studied. A common method employed for computing the join is the nested loops method [Blasgen77, Selinger79]. The nested loops method can be made more efficient by using an index on the joining column of the inner relation. Another popular method for the join is the merge-scan method. [Blasgen77] and [Yao79] present various algorithms and compare their costs. [Gotlieb75] introduced CPU cost into the optimization model for joins. [Valduriez85] presents the concept of a **join index**, which is a binary relation, for computing joins efficiently. The join of two relations is precomputed and stored in the join index. [Valduriez85] shows that the join index performs very well for joins with low selectivity factors.

Join methods based on hashing techniques [Goodman81, Kitsuregawa83, Valduriez84, DeWitt84, DeWitt85] have become popular for computing joins in a multi-processor environment. There has also been a vast amount of literature studying **semijoins** and their use in executing joins [Bernstein81, Bernstein81a, Bernstein81b]. Semijoins are particularly useful in a distributed environment where communication costs are very high [Bernstein81a]. Semijoins can be used to evaluate **tree-queries** [Bernstein81b].

Queries involving more than one join are still very difficult to optimize. One major problem is the estimation of sizes of intermediate relations. [Ibaraki84] shows that determining an optimal nesting order for n-way joins is NP-hard. [Krishnamurthy86] develops a quadratic algorithm for determining a join order that is close to the optimal join order for n-way joins. However, both of these papers assume that join column values are uniformly distributed and that join selectivities are known for all joins.

There has also been research in the areas of global query optimization and distributed query optimization which is not of relevance here. It is clear from our review of the query optimization literature that little or no work has been focused on queries involving multiple disjuncts. The focus of this thesis is to show how to optimize queries involving multiple disjuncts.

# CHAPTER 3

## MULTIPLE-RELATION QUERIES

In this chapter, we discuss the optimization of multiple-relation queries. The techniques for optimizing single-disjunct single-component queries are well known [Smith75, Wong76, Selinger79, Yao79, Youseffi79]. For the sake of completeness, we begin by first discussing the optimization of single-disjunct multiple-component queries. The rest of the chapter is devoted to the optimization of multiple-relation multiple-disjunct queries. Since optimization techniques for **conjunctive (single disjunct)** queries in relational databases are well known, the natural way to evaluate a multiple-disjunct query was to execute each disjunct independently [Bernstein81, Kerschberg82]. However, evaluating each disjunct independently may be very inefficient. In this chapter, we develop methods that merge two or more disjuncts to form a **term**. The advantage of merging disjuncts to form terms lies in the fact that each term can be evaluated with a single scan of each relation that is present in the term. In addition, the number of times a join is performed will also be reduced when two or more disjuncts are merged into a single term. The criteria for merging a set of disjuncts will be presented. As we will see, the number of times each relation in the query is scanned will be equal to the number of terms. Thus, minimizing the number of terms will minimize the number of scans of each relation. We will formulate the problem of minimizing the number of scans as one of covering a **merge graph** by a minimum number of **complete merge graphs** which are a restricted class of **cartesian product graphs**. In general, the problem of minimizing the number of scans is NP-complete. We present polynomial time algorithms for a special case of merge graphs called **simple merge graphs**. We also present a heuristic for merge graphs that are not simple.

Throughout this chapter, we will assume that no relations have any indices on them and that we are only concerned with reducing the number of scans for each relation present in the query. What about relations that have indices on them? It turns out that our performance metric of reducing the number of scans is beneficial even in the case that there are indices. In the next chapter, we will demonstrate that when optimizing single-relation multiple-disjunct queries, the cost (measured in terms of disk accesses) may be reduced if all the disjuncts are optimized together rather than individually. Thus, our algorithm for

minimizing the number of terms is also very beneficial in cases where indices exist.

In this chapter, we demonstrate that optimizing queries involving multiple disjuncts by optimizing each disjunct separately can be very inefficient. Considerable savings in cost may be achieved by optimizing the disjuncts together.

### 3.1. The Meaning of Arbitrary Queries

Most queries are simple and easy to comprehend. This is because queries normally have only a single disjunct and only a single component. However, the QL of a query can have multiple disjuncts in which each disjunct consists of multiple components. Thus, we need to have a simple way of understanding these queries in order to optimize and run them efficiently. The following interpretation of what such a query means is fairly standard [Wong76, Ullman82].

> Take the cartesian product of all the relations involved. Call it U. Apply the qualification list as a whole to each tuple in U and project out the required columns and eliminate duplicates.

This interpretation is significant because it tells us that a query is valid even though it may be difficult to comprehend its meaning.

### 3.2. Single-Disjunct Multiple-Component Queries

While components[6] enhance the expressive power of QUEL, queries with more than one component become difficult to comprehend. By this we mean that it would be hard to state the intent of such a query in words, and hence hard for people to understand. This is probably why optimization of components has not received much attention. The result of a disjunct is the cartesian product of its components. Given the **adjacency matrix** or any other representation of the query graph, the various components can be isolated using a **depth first search** [Sedgewick84]. A couple of examples of single-disjunct multiple-component queries follow.

**Example:**

retrieve (**s**.sno, **p**.pno)

This query has two components and an implicit cartesian product.

[6]In [Wong76] components correspond to **disjoint subqueries.**

**Example:**

retrieve (s.name) where **p**.pno > 50

This query also has two components as there is no join clause between **s** and **p**. It means "retrieve all supplier names if there exists at least one part whose number is greater than 50". If no such part exists return nothing! In other words the above query has an *if-statement* embedded in it. This is because no attribute of **p** appears in the target list. Another interpretation of the above query involves taking the cartesian product between the **p**.pno column whose values are greater than 50 and the s.name column and finally projecting on the name column[7]. Both interpretations will give identical results but treating the query as an if-statement is more efficient, as the processing of an if-component need proceed only until a single result tuple is obtained. This is because an if-component has no attributes in the TL. If-components are easy to identify using the definition given in Section 1.3.

The following query

```
retrieve (s.name, s.city)
where
   x.id > 50
 and s.sno = p.pno
 and s.city = "Madison"
 and x.a = y.b
```

is equivalent to

```
retrieve (s.name, s.city)
where
   s.sno = p.pno
 and s.city = "Madison"
 if
 (x.id > 50 and x.a = y.b )
```

The relation **p** (parts) does not occur in the target list but is not part of the if-component because it occurs in a join clause where the other relation is present in the target list. One needs to evaluate the if-component only until the first tuple is produced. If no such tuple results, the result of the entire disjunct is empty! This means that no other components of that disjunct need to be evaluated. Thus, it might be a good idea to execute the if-components first. It is not clear if access planning for an if-component, involving multiple relations, should be treated differently from that of an ordinary component. Ideally, we would

[7]Duplicates, if present, may be eliminated from the result.

like to determine the existence of result tuples without completely evaluating the if-component. Consider the following query:

```
retrieve (x.a)
where
   s.sno < 7
 and p.pno < 10
 and s.sno = p.pno
```

Assuming the optimizer knows that the s.sno and p.pno values lie in the range of 1-100 and each of the 100 values is present in both columns, the optimizer can just project out x.a. On the other hand, if the optimizer cannot determine the existence of result tuples, the if-component must be executed. In general, it may be more efficient if we can evaluate the if-component incrementally. In other words, it may not be necessary to get all tuples from the two selections before determining if the join will yield result tuples. Incrementally evaluating an if-component with multiple joins is more difficult.

In the case of a single-relation if-component, using an index seems optimal. In fact, a single-relation if-component, with a single attribute, can be evaluated by just looking at the appropriate index.

### 3.3. Multiple-Disjunct Queries

#### 3.3.1. The Problem

**Definition**: A **term** is an expression of the form[8] $\prod_{i=1}^{n} P_i$, $n > 0$, where each $P_i$ is either a join clause or a boolean expression (in disjunctive normal form) of selection clauses on exactly one relation. □

We will motivate the problem associated with optimizing each disjunct separately with a couple of examples. Throughout this chapter, $S_i$, $T_j$, and $U_k$ will denote single-relation selection clauses on the relations S, T, and U respectively. J will denote a join clause.

To illustrate the effect of optimizing each disjunct separately, consider the following query with four disjuncts[9]:

$$S_1 \cdot T_1 \cdot J + S_1 \cdot T_2 \cdot J + S_2 \cdot T_1 \cdot J + S_2 \cdot T_2 \cdot J \text{ (Example 1)}$$

---

[8] $\prod_{i=1}^{n} P_i$ is equivalent to $P_1$ AND $P_2$ AND ... AND $P_n$.

If the four disjuncts in this query were optimized and run separately, the S and the T relations would each be scanned four times and the join J would be executed four times, each time with different inputs. On the other hand, if the four disjuncts are transformed into the following term:

$$(S_1+S_2)\cdot(T_1+T_2)\cdot J$$

S and T will each be scanned once and J will be executed only once.

As another example, the query,

$$S_1\cdot J + T_2\cdot J + S_2\cdot J + T_1\cdot J \text{ (Example 2)}$$

can be transformed to the equivalent query

$$(S_1+S_2)\cdot J + (T_1+T_2)\cdot J$$

The effect of performing this transformation is to reduce the number of scans of both S and T from 4 to 2 and the number of joins that must be executed from 4 to 2.

### 3.4. Covering by Complete Merge Graphs

In this section, we show how the problem of transforming a query in disjunctive normal form into a form that minimizes the number of terms can be formulated in terms of covering a **merge graph** with a minimum number of **complete merge graphs**. Each complete merge graph will correspond to a term in the result and vice-versa.

**Definition**: Given a query in disjunctive normal form, there is a one to one correspondence between the vertices of the **merge graph** and the disjuncts in the query. An edge of color $\chi$ is drawn between two vertices in the merge graph if and only if the two vertices satisfy each of the following three conditions:

**Condition 1**: The two vertices (disjuncts) have selection clauses on the same set of relations.

**Condition 2**: The two vertices have the same set of join clauses.

**Condition 3**: The two vertices differ in the selection clauses of exactly one relation, namely relation $\chi$. □

A complete merge graph is defined in terms of a cartesian product graph that we will define first.

**Definition**: A **cartesian product** graph $G = G_1 \times G_2$ is defined as follows: the vertex set V(G) is $V(G_1) \times V(G_2)$ and $(x_1, x_2)$ -- $(y_1, y_2)$ exists if and only if $x_2 = y_2$ and $x_1$--$y_1$ is an edge in $G_1$, or $x_1 =$

[9]Following standard boolean notation, we use '+' to denote the boolean OR while '·' is used to denote the boolean AND.

$y_1$ and $x_2$--$y_2$ is an edge in $G_2$. □

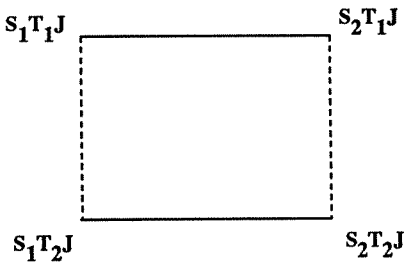
Thus, if $G = G_1 \times G_2 \times ... \times G_m$, each pair $(v_1, ..., v_m)$, $(w_1, ..., w_m)$ of adjacent nodes in G differs on exactly one coordinate.

**Definition:** A **complete merge graph** $G = G_1 \times G_2 \times ... \times G_m$ is a cartesian product graph where each $G_i$ is a clique. □

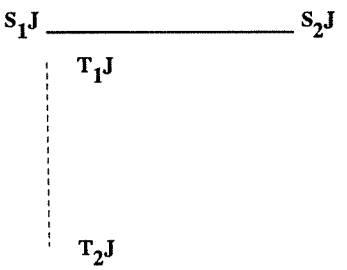
In drawing complete merge graphs, we will adopt the following convention: An edge in a complete merge graph between two adjacent nodes, $(v_1, ..., v_m)$ and $(w_1, ..., w_m)$, has color i where $v_i \neq w_i$ and is drawn parallel to the ith coordinate axis.

Figure 3.1 illustrates the merge graphs for the two examples described in Section 3.3.1. By definition, the disjuncts correspond to the vertices in the merge graphs. The dotted edges connect vertices that differ in selection clauses on relation T only, while the solid edges connect vertices that differ in selection clauses of the S relation only. The merge graph of Example 1 is also a complete merge graph, while the merge graph of Example 2 consists of two complete merge graphs.

Figure 3.2 shows some examples of complete merge graphs and the corresponding terms. Figure 3.2(A) shows a one-dimensional complete merge graph for the query given by $S_1 + S_2 + S_3 + S_4$. Figure 3.2(B) shows a two-dimensional complete merge graph for the query given by $S_1 \cdot T_1 + S_1 \cdot T_2 + S_2 \cdot T_1 + S_2 \cdot T_2 + S_3 \cdot T_1 + S_3 \cdot T_2$. Similarly, Figure 3.2(C) shows a three-dimensional complete merge graph for the




**Merge Graph for Example 1.**
**(Single Component)**




**Merge Graph for Example 2.**
**(Two Components)**

**Figure 3.1**

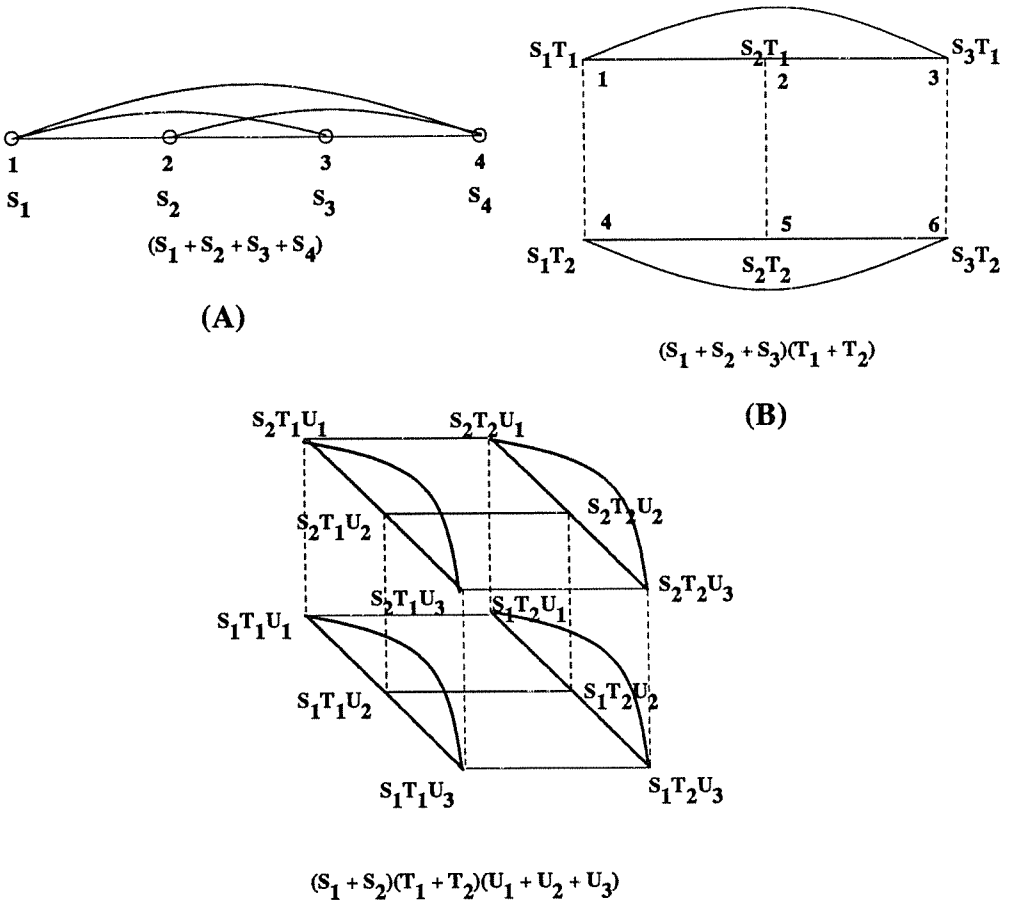



**Figure 3.2**

query with twelve disjuncts. The figures show that the selection clauses of the various relations serve as coordinates in drawing merge graphs. Since all the vertices in a connected component of a merge graph have the same set of join clauses, we choose to drop them from the disjuncts whenever it is convenient to do so. Clearly there is a 1-1 correspondence between a term and a complete merge graph. Given a term $\prod_{i=1}^{n} P_i$, the corresponding complete merge graph is the cartesian product of the complete graphs $G_i$, i = 1, n. If $P_i = p_1 + p_2 + ... + p_{|P_i|}$, the vertex set of the corresponding $G_i$ is $(p_1, p_2, ..., p_{|P_i|})$. Similarly, given a complete merge graph, we can obtain the corresponding term uniquely. For example, given the complete merge graph $(S_1, S_2, S_3) \times (T_1, T_2)$ (shown in Figure 3.2(B)), the corresponding unique term is $(S_1 + S_2 +$

$S_3)\cdot(T_1 + T_2)$.

Thus, given an arbitrary set of disjuncts, the problem of minimizing the number of terms is equivalent to covering the vertices of the corresponding merge graph with a minimum number of complete merge graphs. A vertex is said to be covered if it is part of at least one complete merge graph in the cover. A cover that consists of a minimum number of complete merge graphs is called a **minimum cover**.

### 3.5. Covering by the Minimum Number of Complete Merge Graphs is NP-complete

The problem $P_1$, of finding a minimum cover of complete merge graphs in a merge graph is NP-complete. In fact, this is true even when the corresponding set of disjuncts involve only two relations. The proof is simple [Pruhs87]. We state problem $P_1$ formally.

**Instance**: A two dimensional merge graph G' = (V', E'), positive integer $K \leq |V'|$.

**Question**: Are there $k \leq K$ complete merge graphs that cover G'?

**Theorem 1**: Problem $P_1$ is NP-complete.

**Proof**: It is easy to see that $P_1$ is in NP, since a nondeterministic algorithm need only guess k complete merge graphs and check in polynomial time that the merge graph is covered by them.
It is known that the problem $P_2$, of covering a given bipartite graph with the minimum number of complete bipartite subgraphs is NP-complete [Garey79]. We state problem $P_2$ formally.

**Instance**: Bipartite graph G = (X, E, Y), $(V = X \cup Y)$, positive integer $K \leq |E|$.

**Question**: Are there $k \leq K$ subsets $V_1, V_2, ..., V_k$ of V such that each $V_i$ induces a complete bipartite subgraph of G and such that for each edge $x\text{--}y \in E$ there is some $V_i$ that contains both x and y?
We will polynomially reduce an instance of problem $P_2$ to an instance of problem $P_1$ as follows:
For every edge x--y in G, associate a disjunct (x', y') in the two dimensional merge graph, G'. Therefore, we have $|E| = |V'|$.

We now claim that G can be covered with $k \leq K$ complete bipartite subgraphs if and only if G' can be covered with k complete merge graphs. By the reduction, every point in the merge graph corresponds to an edge in the bipartite graph and vice-versa. Therefore, with every complete bipartite subgraph, $(X_i, E_i, Y_i)$ in G, we can associate the complete merge graph $X'_i \times Y'_i$ in G' in a 1-1 fashion. □

In the following sections, we will develop polynomial time algorithms for special classes of merge graphs. We present a heuristic for general merge graphs in Section 3.8.

### 3.6. A Quadratic Algorithm for Simple Merge Graphs

We begin this section with a couple of definitions.

**Definition**: A **corner vertex** cv in a merge graph G is a vertex such that all edges incident on cv are part of one complete merge graph H. We say that H is rooted at cv and denote it by Hcv. □

**Definition**: A merge graph G is said to be **simple** if $V(G) = V(Hcv_1) \cup V(Hcv_2) \cup ... \cup V(Hcv_n)$. □

We will show that we can cover a simple merge graph $G_s$ with a minimum number of complete merge graphs in $O(|V(G_s)|^2)$ time.

**Definition**: The **dimension** of a vertex v, denoted by dim(v), is equal to the number of distinct colors of the edges incident on v. □

**Definition**: The **degree_vector** of a vertex v, denoted by deg_vec(v), is given by: deg_vec(v) = $(n_1, n_2, ..., n_k)$ where k = dim(v), $n_i$ is the number of edges with color i incident on v, i = 1, ..., k. □

Finding if a vertex v (dim (v) = k, deg_vec(v) = $(n_1, n_2, ..., n_k)$) is a corner vertex can be done in $O(|V(G_s)|)$ time. The vertices of a complete merge graph, rooted at v, form a rectilinearly oriented, complete k-dimensional grid. The coordinate of each vertex in this grid can be represented as a k-tuple. Each component of this k-tuple is the set of selection clauses on a particular relation. The number of vertices adjacent to v is $(n_1 + n_2 + ... + n_k) < |V(G_s)|$. By inspecting the components of the coordinates of the vertices adjacent[10] to v, we can deduce the coordinates of the remaining vertices that must be present in the merge graph in order for v to be a corner vertex. This can be done in $O((n_1 + n_2 + ... + n_k) < |V(G_s)|)$ time. We now need to check if the remaining $(n_1 * n_2 * ... * n_k)$ vertices are present in the merge graph. Either $(n_1 * n_2 * ... * n_k) \leq |V(G_s)|$ or $(n_1 * n_2 * ... * n_k) > |V(G_s)|$. In the latter case we know that there is no complete merge graph rooted at v. Checking for the existence of a vertex with a given coordinate can, in practice, be done in O(1) time by hashing on the coordinate. Only if all the $(n_1 * n_2 * ... * n_k)$ vertices exist, then is v a corner vertex. For example, in Figure 3.2(B), if the input vertex was $S_1 \cdot T_1$, we know that

[10]The merge graph is stored as adjacency lists.

its adjacent vertices are $S_2 \cdot T_1$ and $S_3 \cdot T_1$ along the edges with label S and $S_1 \cdot T_2$ along the edge with label T. Thus the S-components of the coordinates of the $S_2 \cdot T_1$ vertex and the $S_3 \cdot T_1$ vertex are $S_2$ and $S_3$ respectively. Similarly, the T component of the $S_1 \cdot T_2$ vertex is $T_2$. Now all we need to do is to check the presence of vertices with labels $S_2 \cdot T_2$ and $S_3 \cdot T_2$.

We now present the $O(|V(G_s)|^2)$ time algorithm for covering a simple merge graph with the minimum number of complete merge graphs.

**Input**: A simple merge graph $G_s$.

**Output**: The (minimum number of) complete merge graphs that cover $G_s$.

**Algorithm A**
V_I = ∅; /* V_I represents the set of Vertices that have been included in a complete merge graph */
**while** (corner vertices remain in (V - V_I)) **do**
**begin**
    **Step 1**: Find a corner vertex cv such that cv ∈ (V - V_I).
    **Step 2**: Find the complete merge graph $H_{cv}$ rooted at cv. V_I = V_I ∪ V(Hcv).
**end while**
**End Algorithm A**

For reasons of efficiency, we will always search the vertices of a merge graph for a corner vertex in order of increasing degree. The proof of correctness of Algorithm A is presented in the following theorem.

**Theorem 2**: At the end of Algorithm A, V_I = V, and Algorithm A produces the minimum cover.

**Proof**: The proof is simple and is based on the following two observations:

1. Every corner vertex belongs to one and only one complete merge graph.
2. Two corner vertices with different corresponding complete merge graphs cannot simultaneously be part of any one particular complete merge graph.

The first observation follows directly from the definition of a corner vertex. All the edges adjacent to a corner vertex belong to the complete merge graph rooted at that corner vertex. Therefore, every corner vertex can belong to one and only one complete merge graph.

We will prove the second observation by contradiction. Assume that the two different complete merge graphs are $Hcv_1$ and $Hcv_2$. Assume that $cv_1$ and $cv_2$ are part of a third complete merge graph $Hcv_3$. By the first observation, $cv_1$ can belong to only one complete merge graph. This implies that $Hcv_1$ is identical to $Hcv_3$. Similarly, $Hcv_2$ is identical to $Hcv_3$. Therefore $Hcv_1$ is the same as $Hcv_2$.

The number of complete merge graphs in the minimum cover is equal to the number of iterations of the while loop in Algorithm A. This is because at each iteration exactly one complete merge graph is obtained. Hence the theorem. □

### 3.7. An Improved Algorithm for a Larger Class of Merge Graphs

Algorithm A assumes that a corner vertex would be found at every iteration and works optimally only for simple merge graphs. Figure 3.3(A) shows a merge graph G that is not simple. Vertices 7 and 8 are the only corner vertices[11] in G and $V(H_7) \cup V(H_8) \neq V(G)$. After finding corner vertex 7, if we had removed the vertices of the corresponding complete merge graph and edges adjacent to them, both vertices 1 and 4 would have become corner vertices. However, in general, after finding a corner vertex, we cannot remove all the vertices in the corresponding complete merge graph along with the adjacent edges without affecting the optimality of the result. The merge graph in Figure 3.3(B) can be reduced to two terms, viz., $(S_1 + S_2)\cdot(T_1 + T_2)\cdot U_1$ and $S_2\cdot T_2\cdot U_2$. On the other hand, if vertex 5 is identified first as a corner vertex and we remove vertices 3 and 5, we would finally get three result terms, viz., $S_2\cdot T_2\cdot(U_1 + U_2)$, $(S_1 +$

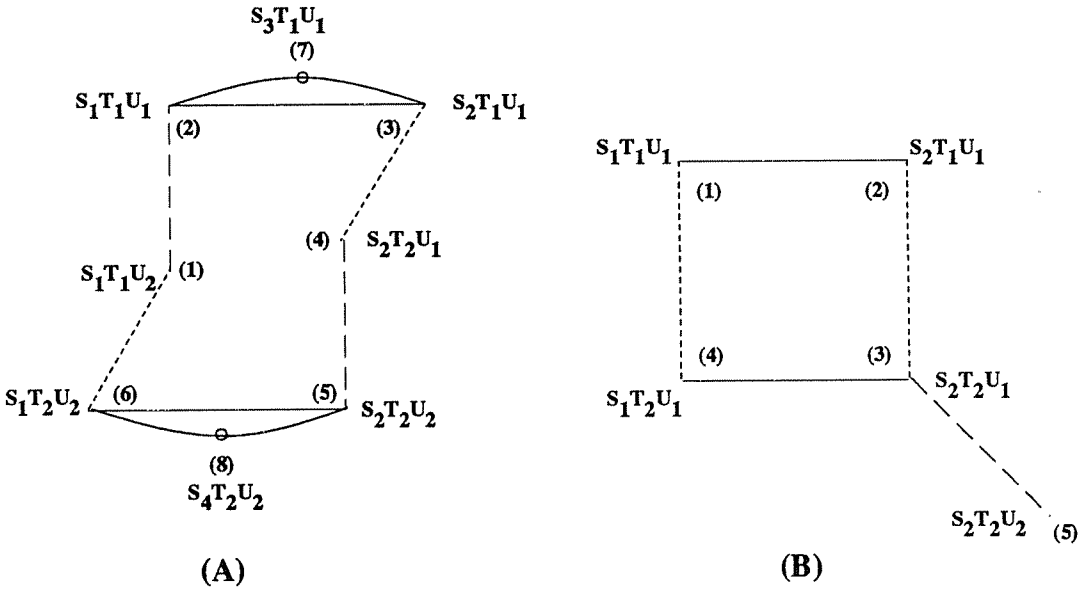



**Figure 3.3**

[11]Vertex 4 would have been a corner vertex if the disjunct $S_2\cdot T_1\cdot U_2$ was in the query. Similarly, vertex 3 would have been a corner vertex if the disjuncts $S_1\cdot T_2\cdot U_1$ and $S_3\cdot T_2\cdot U_1$ were in the query.

$S_2 \cdot T_1 \cdot U_1$, and $S_1 \cdot T_2 \cdot U_1$. Clearly, that would not be optimal.

Let $Hcv_1$ be a complete merge graph that was identified in some iteration of Algorithm A. Before proceeding to the next iteration, we would like to:

(1) Remove certain qualifying edges in $E_1$, $E_1 = \{e = v\text{--}u \mid v \in V(Hcv_1)\}$, and

(2) Remove certain qualifying vertices from $V(Hcv_1)$.

Therefore, we need some criteria that must be met by the qualifying edges and vertices before they can be deleted. The criteria must ensure that the optimality of the result is not affected after qualifying edges and vertices are deleted. We must ensure that every vertex that was a corner before isolating $Hcv_1$ remains a corner vertex after removing the qualifying edges and vertices. We will first present the criteria and then an algorithm that we will call Algorithm B. The class of merge graphs for which Algorithm B will work optimally is the class of merge graphs in which a corner vertex will be found at every iteration after qualifying edges and vertices are removed. This class includes the class of simple merge graphs.

Let $Hcv_1$ be a complete merge graph identified at some iteration. Let $Hcv_2 = V_1 \times V_2 \times \ldots \times V_n$ be some other complete merge graph such that $C = V(Hcv_1) \cap V(Hcv_2)$ is a non-empty set. The subgraph induced by the vertices of C, denoted by G[C], is a complete merge graph. This is because complete merge graphs are cartesian product graphs and the intersection of two or more cartesian product graphs is also a cartesian product graph. $G[C] = W_1 \times W_2 \times \ldots \times W_n$, where $W_i \subseteq V_i$, i = 1, n. We define three boolean functions **Delete_Edge(e)**: $E_1$ --> {false, true}, **Delete_Vertex(v)**: $V(Hcv_1)$ --> {false, true}, and **Undelete_Edge($v_a$, $v_b$)**: $V(Hcv_1) \times V(Hcv_1)$ --> {false, true} which are true only in the following cases:

For $v \in V(Hcv_1)$ and e = v--u,

If $e \in E(Hcv_1)$, Delete_Edge(e) is true.

If $e \notin E(Hcv_1)$, then Delete_Edge(e) is true if and only if there is no edge e' adjacent to v, $e' \notin E(Hcv_1)$, such that e and e' are the adjacent edges of the same chordless four cycle. Clearly the adjacent edges of a chordless four cycle will be of different colors.

Delete_Vertex(v) is true if Delete_Edge(e) is true for every edge e adjacent to v. Clearly, Delete_Vertex(v) is true if $dim'(v) \leq 1$. dim'(v) is calculated using edges that do not belong to $E(Hcv_1)$.

Undelete_Edge($v_a$, $v_b$) is true if Delete_Vertex($v_a$) and Delete_Vertex($v_b$) are both false, where $v_a$, $v_b \in V(Hcv_1)$.

**Theorem 3**: Let $Hcv_1$ and $Hcv_2$ be two complete merge graphs as described above. After

(1) removing all edges e such that Delete_Edge(e),
(2) removing all vertices v such that Delete_Vertex(v), and
(3) undeleting (putting back) all edges $v_a$--$v_b$ such that Undelete_Edge ($v_a$, $v_b$)

$cv_2$ will still be a corner vertex of a complete merge graph $H'cv_2$ and $V(Hcv_1) \cup V(Hcv_2) = V(Hcv_1) \cup V(H'cv_2)$.

**Proof**: All edges in $E(Hcv_1)$ are first dropped. We divide the proof into two cases:

**Case 1**: $W_k \subset V_k$ for some k, $1 \leq k \leq n$; $W_j = V_j$, $j \neq k$, $1 \leq j \leq n$.

Consider $v \in C$. Every edge adjacent to $v \notin E(Hcv_1)$ as edges in $E(Hcv_1)$ have already been removed. Since $cv_2 \notin C$, we have $cv_2 \in V_1 \times V_2 \times \ldots \times (V_k - W_k) \times V_{k+1} \times \ldots \times V_n$. Assume that there is more than one edge incident on v. Delete_Edge($e_i$) is true for every edge of the form $e_i = v\text{--}cv_2$ as it will be adjacent to edge $v\text{--}v_1$, $v_1 \in V(Hcv_2)$ such that $color(v\text{--}cv_2) = color(v\text{--}v_1) = k$. If $v_1 \notin V(Hcv_2)$, then edge $v\text{--}cv_2$ and edge $v\text{--}v_1$ cannot be edges of the same chordless four cycle as this will contradict the fact that $cv_2$ is a corner vertex. Thus, all vertices in $V_1 \times V_2 \times \ldots \times (V_k - W_k) \times V_{k+1} \times \ldots \times V_n$ that were corner vertices before will remain corner vertices. The complete merge graph, $H'cv_2$, rooted at $cv_2$ is given by $V_1 \times V_2 \times \ldots \times (V_k - W_k) \times V_{k+1} \times \ldots \times V_n$. Clearly, $V(Hcv_1) \cup V(Hcv_2) = V(Hcv_1) \cup V(H'cv_2)$.

**Case 2**: Let $T = \{k, m, \ldots\}$. $W_i \subset V_i$, $i \in T$; $W_j = V_j$, $j \notin T$.

In other words, more than one W is a proper subset of the corresponding V. Consider $v \in C$. There exist edges $e_k$, $e_m \notin E(H'cv_1)$ such that $e_k = v\text{--}v_k$, $v_k \in V_1 \times V_2 \times \ldots \times (V_k - W_k) \times V_{k+1} \times \ldots \times V_n$. and $e_m = v\text{--}v_m$, $v_m \in V_1 \times V_2 \times \ldots \times (V_m - W_m) \times V_{m+1} \times \ldots \times V_n$. Clearly, edges $e_k$ and $e_m$ are the adjacent edges of a chordless four cycle as they belong to the complete merge graph, rooted at $cv_2$. In this case, none of the vertices in $Hcv_2$ will be deleted and $cv_2$ will still be a corner vertex. The complete merge graph, $H'cv_2$, is the same as $Hcv_2$. Again, $V(Hcv_1) \cup V(Hcv_2) = V(Hcv_1) \cup V(H'cv_2)$. □

The above theorem applies to any pair of 'intersecting' complete merge graphs that are in turn embedded in a larger merge graph. We now present an augmented version of Algorithm A which we will call Algorithm B. Algorithm B removes qualifying edges and vertices.

**Input**: A merge graph G.

**Output**: A set of complete merge graphs that cover a subset of V(G).

**Algorithm B**
V_I = ∅; Orig_vertex_set = V;
**while**(corner vertices remain in (V - V_I))**do**
**begin**
**Step 1**: Find a corner vertex $cv_1$, $cv_1 \in$ (V - V_I). /* If dim(v) = 1, v is a corner vertex by definition. */
**Step 2**: Find the complete merge graph $Hcv_1$ rooted at $cv_1$. V_I = V_I ∪ V($Hcv_1$).
**Step 3**: Delete edge e = v--u if Delete_Edge(e) is true, v ∈ V($Hcv_1$).
**Step 4**: Delete v ∈ V($Hcv_1$) if Delete_Vertex(v) is true[12].
*Step 5: Undelete all edges $v_a$--$v_b$ ($v_a$, $v_b$ ∈ V($Hcv_1$)) such that Undelete_Edge ($v_a$, $v_b$) is true.*
**end while**
**End Algorithm B**

Step 5 has been introduced strictly for reasons of correctness. We will elaborate on this point at the end of the section.

If, at the end of the algorithm, V_I = Orig_vertex_set, by Theorems 2 and 3, we know that the algorithm has produced the optimal number of terms.

A note on the complexity of Algorithm B. Steps 1 and 2 are identical in algorithms A and B. Step 3, the step with the greatest time complexity, determines if an edge is part of a chordless four cycle. Checking if two adjacent edges are part of a chordless four cycle can be done in O(1) time (by hashing) by checking for the presence of the fourth vertex. A given edge is adjacent to at most |E(G)| - 1 edges of the merge graph. Thus, the time complexity of Step 3 in Algorithm B is $O(|E(G)|^2)$. The number of times the while loop is executed is at most O(|V(G)|). Therefore, the time complexity of Algorithm B is $O(|V(G)| * |E(G)|^2)$.

We will conclude this section with an example that illustrates the operation of Algorithm B.

**Example**: Consider the merge graph shown in Figure 3.4(A). Each vertex in the merge graph represents the disjunct that is a product of its coordinates. For example, vertex 1 denotes the disjunct $S_1T_4U_1$ while vertex 12 denotes $S_2T_1U_2$. There are a total of four corner vertices in the merge graph. They are {1, 10, 11, 12}. Assuming $cv_1$ = 10, we can see that the set of vertices in the complete merge graph $H_{10}$ is = {4, 5, 7, 8, 9, 10}. The vertices in V($H_{10}$) can be merged into the single term $(S_2+S_3)\cdot(T_1+T_2+T_3)\cdot U_1$. Notice that $H_{10}$ intersects with $H_{11}$ as described in case 1 (in the proof) while $H_{10}$ intersects with $H_1$ as described in case 2. The set of edges adjacent to vertices in V($H_{10}$) is $E_1$ = {4-

[12]It is possible that G may have broken up into more than one component after step 4. This can happen if more than one vertex of $Hcv_1$ is an articulation point of G. If G has more than one component, each component must be dealt with individually.

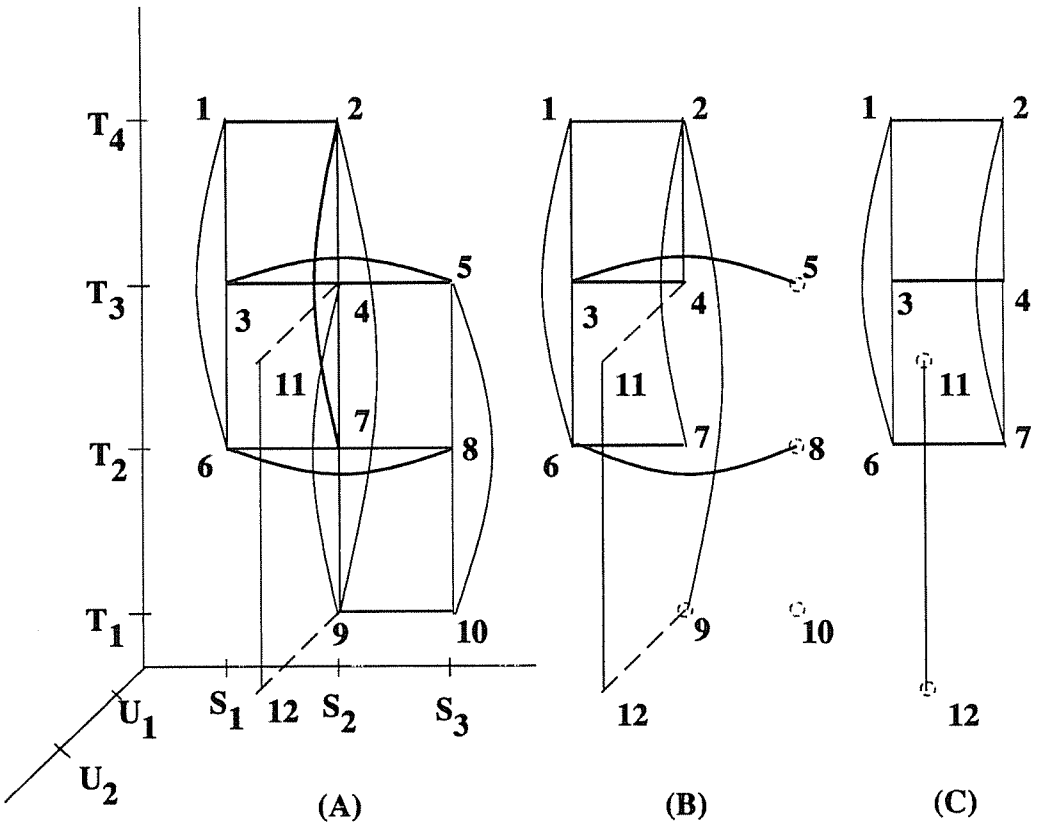



**Figure 3.4**

-2, 4--3, 4--5, 4--7, 4--9, 4--11, 5--3, 5--8, 5--10, 7--2, 7--6, 7--8, 7--9, 8--6, 8--10, 9--2, 9--10, 9--12}. Figure 3.4(B) shows the merge graph after the edges in $E(H_{10})$ have been deleted. We calculate dim'(v) for v $\in V(H_{10})$. We find dim'(10) = 0; dim'(5) = dim'(8) = 1; dim'(9) = dim'(7) = 2; dim'(4) = 3. We remove vertices 5, 8, 10 and the incident edges 3--5, 6--8. The set of edges adjacent to vertices in $V(H_{10})$ is now $E_1$ = {4--2, 4--3, 4--11, 7--2, 7--6, 9--2, 9--12}. Of these, edges 4--11, 9--12, and 9--2 can be removed as they are not part of any chordless four cycle. Since vertices 4 and 7 are not removed, we put back edge 4--7. Figure 3.4(C) shows the 2 component merge graph at this stage. Notice that vertices 1, 11, and 12 are still corner vertices.

In the next two iterations, $H_1$ and $H_{11}$ will be found. The algorithm terminates after the third iteration. □

We pointed out earlier that Step 5 was introduced strictly for reasons of correctness. Step 5 caused edge 4--7 to be put back into the merge graph in the above example. By definition of a complete merge graph, $H_1$ would not be a complete merge graph without the edge 4--7. However, as described in section 3.6, we would still have been able to recognize that vertex 1 is a corner vertex because the edges adjacent to vertex 1 were not removed. In order to determine if a set of vertices are the vertices of a complete merge graph rooted at a given vertex v, our algorithm requires that only those edges adjacent to v be present along with all the vertices of the complete merge graph. The presence of the edges that are not adjacent to a corner vertex but that belong to the corresponding complete merge graph need not be present for finding all the vertices of the complete merge graph and the associated term.

### 3.8. Maximum and Maximal Merge Graphs

Algorithm B assumed that a corner vertex would be found at every iteration. Unfortunately, it is easy to find merge graphs that have no corner vertices. Figure 3.5 shows an example of a merge graph with no corner vertex that can be covered optimally with six complete merge graphs as shown. In the absence of corner vertices, it seems natural to start by finding a maximum merge graph rooted at some vertex.

**Definition**: Let E be the edges adjacent to a vertex v. A **maximum (maximal) merge graph** H' = (V', E') rooted at a vertex v, $E' \subseteq E$, is defined to be a complete merge graph rooted at v such that the set of the vertices, V', of the complete merge graph form a maximum (maximal[13]) set. □

However, finding a maximum merge graph rooted at a vertex is NP-complete. In fact, the problem $P_1$ of finding a maximum merge graph rooted at any vertex in the (2-dimensional) merge graph is NP-complete. Formally, the problem $P_1$ may be stated as follows:

**Instance**: Given a two dimensional merge graph G' = (V', E') and a positive integer $M = K*K \leq |V'|$.

**Question**: Is there a complete merge graph with $\geq M$ vertices in the given merge graph?

**Theorem 4**: Problem $P_1$ is NP-complete.

**Proof**: Clearly $P_1$ is in NP, as a nondeterministic algorithm need only pick $m \geq M$ vertices and

[13]The set of vertices in V' form a maximal set if no more vertices in the merge graph can be added to V' to find another complete merge graph. A maximum set is the largest maximal set.

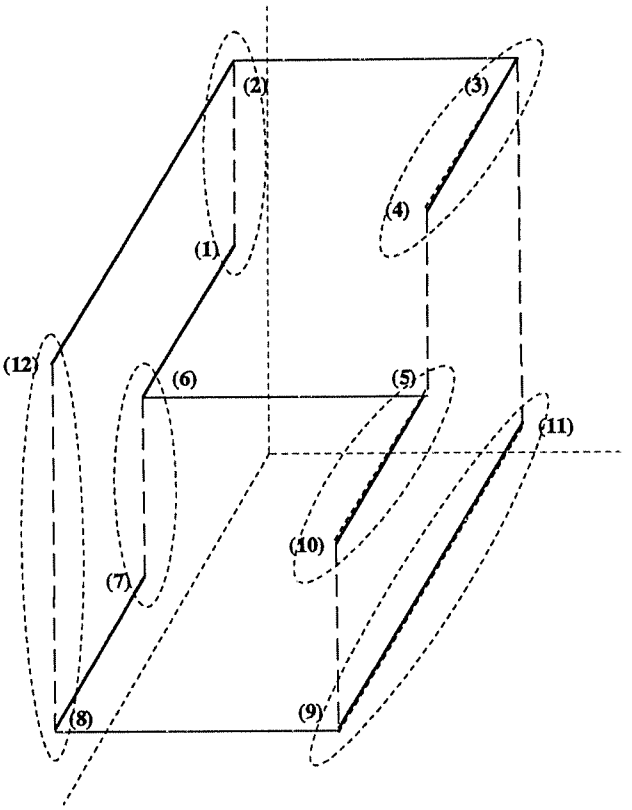



**Figure 3.5**

verify in polynomial time that these m vertices form a complete merge graph. The rest of the proof that $P_1$ is NP-complete involves two reductions and is presented below. We start with the following NP-complete problem in [Garey79].

**Instance**: Graph G = (V, E), positive integer $K \leq |V|$.

**Question**: Does G contain a clique of size K or more, i.e., a subset $V' \subseteq V$ with $|V'| \geq K$ such that every two vertices in V' are joined by an edge in E?

We polynomially reduce G to a bipartite graph $B(V_1, E_B, V_2)$ as follows:

(1) For every vertex i in V, introduce two vertices $i_1 \in V_1$ and $i_2 \in V_2$, and the corresponding edge $i_1$--$i_2$ $\in E_B$.

(2) For every edge i--j $\in$ E, add edges $i_1$--$j_2$, $i_2$--$j_1 \in E_B$.

We now claim that G has a clique of size $\geq$ K if and only if B has a complete bipartite sub-graph of order $\geq$ K induced by the vertex partitions $V'_1 \subseteq V_1$ and $V'_2 \subseteq V_2$ where $|V'_1| = |V'_2| \geq K$. By

the reduction, a clique of size k in G results in a complete bipartite subgraph of order k in B. This is because, there is a one to one correspondence between a clique in G and a complete bipartite graph in B. The other polynomial reduction consists of reducing an instance of a bipartite graph B to a merge graph G' (as shown in Section 3.5). By this reduction we have $|E_B| = |V'|$. Therefore, B has a complete bipartite subgraph of order $\geq K$ if and only if the merge graph has in it a complete merge graph with $\geq K*K$ number of vertices. This is because there is a 1-1 correspondence between an edge in B and a vertex in G'. □

Unfortunately, finding a maximum merge graph rooted at a vertex does not guarantee an optimal solution. As an example consider the merge graph in three dimensions with 12 vertices shown in Figure 3.5. The optimal solution consists of a cover with six complete merge graphs as shown. There are no chordless four cycles in this merge graph. Vertex 2 has three edges incident on it, each of which is a maximum merge graph. If we start with edge 2--3 as the first complete merge graph, Algorithm B will yield a cover that consists of seven complete merge graphs. However, if we start either with edge 2--12 or 2--1, Algorithm B will yield an optimal cover.

Since the problem of finding a maximum merge graph is NP-complete, and does not necessarily lead to an optimal solution, we propose the following heuristic:

> In the absence of corner vertices in step 1 of Algorithm B, choose a maximal merge graph rooted at a vertex of minimum dimension. If two or more vertices have the minimum dimension, find a maximal merge graph rooted at a vertex with the smallest degree.

The complexity of finding a maximal merge graph, rooted at a vertex v (in a merge graph G), with degree_vector(v) = $(n_1, n_2, ..., n_k)$ is $O((n_1 * n_2 * ... * n_k))$. At the present time, we have not been able to derive any analytical bound on the performance of the heuristic. However, it is very likely that after a maximal merge graph has been found, and the qualifying vertices and edges in this maximal merge graph have been removed, there will be corner vertices in the subsequent iterations. On small examples, we have found that the heuristic gives a solution that is close to the optimal solution. As discussed above, the heuristic performs optimally for the merge graph in Figure 3.5. It seems difficult to be able to derive quantitatively how well the heuristic performs in general. The proof of NP-completeness for the problem of finding a minimum cover (in 2-dimensions) shows that the problem is equivalent to covering the edges of an arbitrary bipartite graph with complete bipartite graphs. There seem to be no good approximation algorithms in literature for the latter problem. Empirical methods for judging the quality of the heuristic also

seem infeasible. Consider for example that we generate all possible merge graphs in three dimensions, with four vertices in each direction. The number of such complete merge graphs is equal to $2^{4*4*4} = 2^{64}$.

# CHAPTER 4

## SINGLE RELATION QUERIES

[Selinger79] provides a very comprehensive view of the optimization techniques for single-relation single-disjunct queries. For the sake of completeness, we begin this chapter by briefly describing these techniques. In the remainder of the chapter, we will then focus our attention on the optimization of multiple-disjunct queries. In Chapter 3, we saw how a multiple-relation, multiple-disjunct query can be reduced to a set of terms such that the single-relation expressions in each term were in disjunctive normal form. This form will be a good starting point for the optimization of single-relation, multiple-disjunct queries. The solution space of a single-relation query with selection clauses on m attributes will consist of rectilinearly oriented m-dimensional hyperboxes. Each disjunct in the query will correspond to one hyper-box and vice-versa. Since optimizing the disjuncts independently of each other can be very inefficient, our objective will be to cover the solution space by a set of subspaces, such that the sum of the costs of these subspaces is minimized. Assuming single-attribute indices, these subspaces are bounded above and below along one attribute (dimension) and unbounded along all other dimensions. The width of a subspace, along the bounded attribute, is equal to the range of the attribute values that tuples accessed via the index on that attribute will have. Each subspace has a cost associated with it which is equal to the number of pages fetched via the index on the attribute along which the subspace is bounded.

Covering the solution space with subspaces of minimum cost can be formulated as the problem of finding a minimum cost vertex cover in a restricted class of hypergraphs. However, the problem of finding a minimum cost vertex cover in a hypergraph is, in general, NP-complete. We will present a linear approximation algorithm from the literature that guarantees a solution that is at most $m * W_{opt}$ where $W_{opt}$ is the cost of the minimum vertex cover and m is the number of attributes in the query that have indices built on them.

Finally, we extend our optimization model to include optimization with multi-attribute indices.

### 4.1. Single-Disjunct Queries

To determine the access path for a relation, it is necessary to know the range of each attribute that is present in the query. These ranges can be calculated by transforming a query to the following standard form[14]:

retrieve (TL)
where
$(attr_1 \geq A_1$ AND $attr_1 \leq B_1)$ AND
$(attr_2 \geq A_2$ AND $attr_2 \leq B_2)$ AND
...
...
$(attr_m \geq A_m$ AND $attr_m \leq B_m)$

($Attr_i$ stands for the ith attribute that is present in the query. If the original query specified $attr_i \geq A_i$, the range of $attr_i$ is $[A_i, MAX_i]$ where $MAX_i$ denotes the maximum value in the domain of $attr_i$). In a single-disjunct query, computing the range of every attribute is easy since the range cannot become larger as a result of additional selection predicates, as there are only ANDs in the qualification. In addition, the range will be continuous if there are no $\neq$ relational operators in the selection clauses. We can ignore splitting the range in the presence of $\neq$'s as the $\neq$ relational operators do not significantly affect the optimization process. In light of this, we do not include predicates with $\neq$ operators in the standard form. Such predicates may be applied to filter tuples that satisfy QL in the standard form. Thus, the solution space of a single-disjunct query involving m attributes is a rectilinearly oriented m-dimensional hyperbox. By maintaining histograms on the values of each attribute column, the selectivity factor for each attribute (for $attr_i$, the fraction of tuples of the relation satisfying $A_i \leq attr_i \leq B_i$) and the number of pages that have to be fetched to access tuples that satisfy $A_i \leq attr_i \leq B_i$) can be estimated. Accessing a relation via an attribute that has a clustered index built on it should result in only a subset of the pages of the relation being fetched. The optimal access path would be that index which results in the smallest number of page accesses. [Selinger79] provides different cost formulas depending on whether the index is clustered or not. Tuples retrieved via the optimal index are then tested to see if they satisfy QL. The selectivity factor of a single disjunct QL is equal to the product of the selectivity factors of the individual selection operators in the query (assuming that the values in the various columns are independent of each other). In short, optimization of single-relation single-disjunct queries, after making certain simplifying assumptions, has been

[14]The order in which the attributes are numbered is not important.

extensively studied and the techniques are found in many commercial optimizers.

### 4.2. Multiple-Disjunct Queries

We will transform every multiple-disjunct query into a standard form that we describe below. In a multiple-disjunct query, each disjunct may have a different set of attributes. Each disjunct may be reduced to the standard form as described in Section 4.1. In addition, each disjunct must be augmented with clauses of the form $MIN_i \leq attr_i \leq MAX_i$ for every attribute i that is not present in the disjunct but is present in the query. $MIN_i$ and $MAX_i$ represent the minimum and maximum values, respectively, in the domain of the ith attribute. Now each disjunct will be a rectilinearly oriented m-dimensional hyperbox where m is the total number of attributes in the query. Henceforth we will assume that every multiple disjunct query is already in this standard form.

For the purposes of our examples, we will use a relation **s** with attributes **a** and **b** with indices on both attributes. We also assume that we have available to us a function

number_pages (relation-name, attribute-name, lower-bound, upper-bound)

that will return the number of pages (index and relation pages) that have to be fetched from the disk in order to access all the tuples in the specified relation whose attribute values, for the given attribute, are in the range [lower-bound, upper-bound]. Obviously, if there is no index on the given attribute, the function number_pages will return the number of pages in the entire relation. Throughout this section we will assume that every disjunct in DNF(QL) has at least one indexed attribute in it. Otherwise, the optimal access path for the query would be a file scan of the relation where the result tuples will be those that satisfy QL.

### 4.3. Calculating Selectivity Factors of Arbitrary Boolean Expressions

The single-relation query may be part of a multiple-relation query. In such an event, the result tuples from the single-relation query feed a join. In order to determine the order of joins and the inner and outer relations in each join, it is important to estimate the number of tuples of a relation that will satisfy a given QL. This is fairly simple if there are only ANDs in the QL. However, as the expression becomes complex, determining the selectivity factor also becomes difficult. The algorithm that we will employ is a fairly standard one in probability and statistics. If $\alpha$ and $\beta$ are two selection clauses the following equations hold:

$$SF(\alpha \text{ OR } \beta) = SF(\alpha) + SF(\beta) - SF(\alpha \text{ AND } \beta) \quad (1)$$

$$SF(\alpha \text{ AND } \beta) = SF(\alpha) * SF(\beta) \quad (2)$$

provided $\alpha$ and $\beta$ are independent of each other. (1) can be generalized as follows:

$$SF(\alpha_1 \text{ OR } \alpha_2 \text{ OR } \cdots \text{ OR } \alpha_n)$$
$$= \sum_{i=1}^{i=m} SF(\alpha_i)$$
$$- \sum_{i=1}^{i=m} \sum_{i<j} SF(\alpha_i)*SF(\alpha_j)$$
$$+ \sum_{i=1}^{i=m} \sum_{i<j} \sum_{j<k} SF(\alpha_i)*SF(\alpha_j)*SF(\alpha_k)$$
...

...

$$+ (-1)^{m+1} SF(\alpha_1) * SF(\alpha_2) * \cdots * SF(\alpha_m) \quad (3)$$

Consider the following query

retrieve (TL)
where
($s.a \geq A_1$ --- (a)
OR
$s.b \leq B_1$) --- (b)
AND
$s.a \leq A_2$ --- (c)

Let us now derive the selectivity factor for the above query.

$$SF((a \text{ OR } b) \text{ AND } c)$$
$$= SF((a \text{ AND } c) \text{ OR } (b \text{ AND } c)) \quad \text{(DNF(QL))}$$
$$= SF(a \text{ AND } c) + SF(b \text{ AND } c) - SF(a \text{ AND } c \text{ AND } b \text{ AND } c) \quad \text{(By(1))}$$
$$= SF(a \text{ AND } c) + SF(b \text{ AND } c) - SF(a \text{ AND } c \text{ AND } b)$$
$$= SF(a \text{ AND } c) + SF(b) * SF(c) - SF(a \text{ AND } c) * SF(b) \quad \text{(By(2))}$$
$$= SF(s.a \geq A_1 \text{ AND } s.a \leq A_2)$$
$$+ SF(s.b \leq B_1) * SF(s.a \leq A_2)$$
$$- SF(s.a \geq A_1 \text{ AND } s.a \leq A_2) * SF(s.b \leq B_2)$$

After generating DNF (QL), we can use (3) and then (2) repeatedly to derive the selectivity factor of any

complex boolean expression. In the above example we have assumed that values in different attributes are independent of each other. Using inverted histograms (equal height as opposed to equal width) [Shapiro84], we should be able to determine SF ($A \leq attr \leq B$) to an arbitrary degree of accuracy depending on how small we are willing to make the height of the histogram.

### 4.4. Optimizing Multiple-Disjunct Queries

To illustrate the optimization of single-relation, multiple-disjunct queries, consider the following example of a two disjunct query in standard form.

**Example:**

retrieve (TL)
where
($\mathbf{s.a} \geq A_1$ AND $\mathbf{s.a} \leq A_4$ AND $\mathbf{s.b} \geq B_2$ AND $\mathbf{s.b} \leq B_3$) --- ($D_1$)
OR
($\mathbf{s.a} \geq A_2$ AND $\mathbf{s.a} \leq A_3$ AND $\mathbf{s.b} \geq B_1$ AND $\mathbf{s.b} \leq B_4$) --- ($D_2$)

The solution space, shown graphically in Figure 4.1, consists of two overlapping rectangles (ABCD and EFGH) corresponding to the two disjuncts. As discussed in Section 4.1, each disjunct will give rise to a

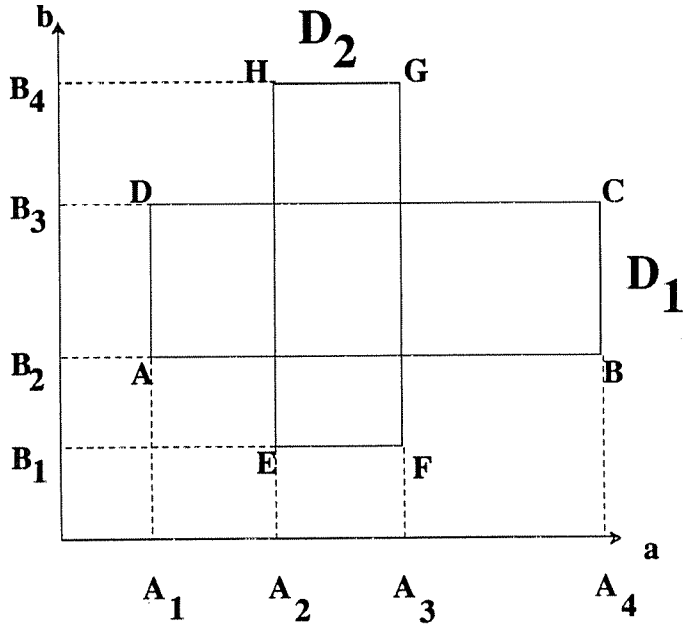



**Figure 4.1**

hyperbox which is a rectangle in the case of two attributes.

[Dayal87] suggests that a query with disjunctions on a single relation be processed with one scan of the relation. This strategy can turn out to be very expensive if indices on the referenced attributes exist. We examined a number of methods to generate access plans for a single-relation multiple-disjunct query. Some of them resulted in very inefficient plans. We shall discuss them in the following sub-sections.

### 4.4.1. The DNF Method

The simplest method consists of optimizing and executing each disjunct independently. The result of the query would be the union of the results of each disjunct. When each disjunct is optimized independently, more pages may be fetched than necessary. For example, assume that there is a clustered index on attribute **a**. Disjunct $D_1$ may have been optimized using the index on **a**. This means that $p_1$ pages are fetched, where $p_1$ = number_pages(**s**, **a**, $A_1$, $A_4$). If disjunct $D_2$ was also optimized using the index on **a** then $p_2$ pages would have been fetched, where $p_2$ = number_pages(**s**, **a**, $A_2$, $A_3$). Clearly $p_2 \leq p_1$ and the set of $p_2$ pages accessed for $D_2$ are contained in the set of $p_1$ pages accessed for $D_1$. Also, since the two rectangles overlap, the results of the two disjuncts may have common tuples. If duplicates are not desired in the result, the cost of duplicate elimination must be incurred.

### 4.4.2. The Disjoint Boxes Method

This method consists of optimizing non-overlapping, or disjoint, hyperboxes separately. As seen in Figure 4.2, the solution space for our 2-disjunct example query consists of the five rectangles, 1 through 5, which are defined by the following expressions:

$\text{rectangle}_1$ : **s.a** $\geq A_1$ AND **s.a** $< A_2$ AND **s.b** $\geq B_2$ AND **s.b** $\leq B_3$
$\text{rectangle}_2$ : **s.a** $\geq A_2$ AND **s.a** $\leq A_3$ AND **s.b** $\geq B_2$ AND **s.b** $\leq B_3$
$\text{rectangle}_3$ : **s.a** $> A_3$ AND **s.a** $\leq A_4$ AND **s.b** $\geq B_2$ AND **s.b** $\leq B_3$
$\text{rectangle}_4$ : **s.a** $\geq A_2$ AND **s.a** $\leq A_3$ AND **s.b** $> B_3$ AND **s.b** $\leq B_4$
$\text{rectangle}_5$ : **s.a** $\geq A_2$ AND **s.a** $\leq A_3$ AND **s.b** $\geq B_1$ AND **s.b** $< B_2$

**Lemma**: If any two boxes in the solution space are adjacent, then the cost (in terms of the number of pages fetched) of the boxes together cannot be greater than the sum of the costs of the two individual boxes.

**Proof**: We will prove the lemma for two dimensions and the proof trivially extends to boxes of more than two dimensions. Figure 4.3 shows two adjacent boxes that can be merged into one box along the horizontal direction. Let the cost of either of the boxes in the vertical direction be z pages. In other words, the

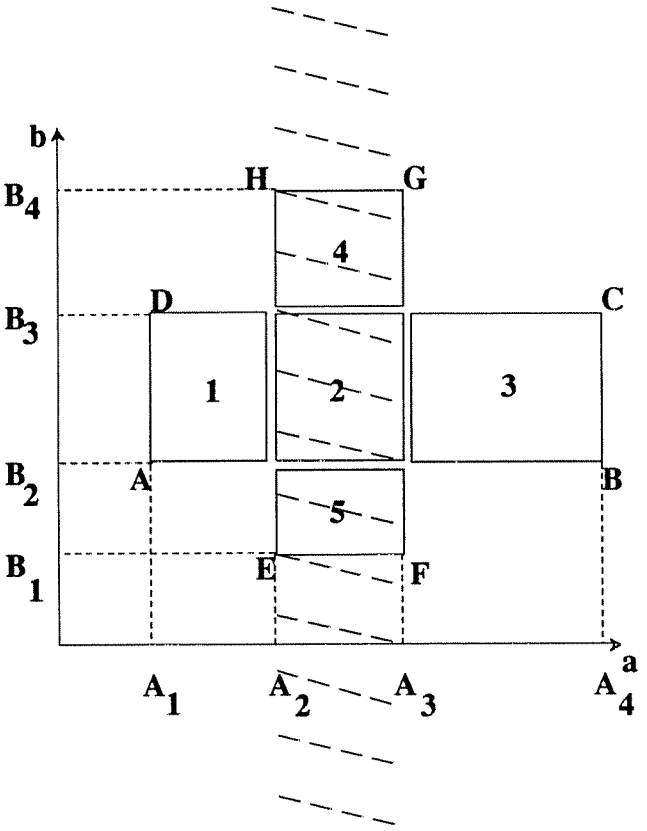
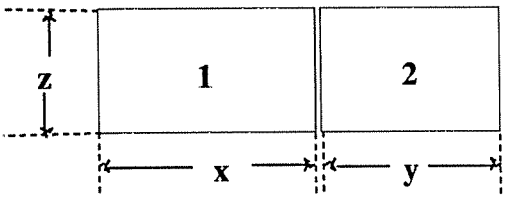



Figure 4.2




$$\min(x+y, z) <= \min(x, z) + \min(y, z)$$

Figure 4.3

cost of using the index on the vertical attribute to retrieve tuples in either of the boxes would be z pages. Similarly, let the costs of boxes 1 and 2 in the horizontal direction be x and y pages respectively. Therefore, the sum of the costs of optimizing the two individual boxes would be given by

COST (Individual boxes) = min (x, z) + min (y, z).

The cost of the two boxes when optimized together is given by

COST (Together) = min (x + y, z).

For any positive values of x, y, z, we have

COST (Together) ≤ COST (Individual). □

From the above lemma it follows that if more than two boxes can be merged along the same direction, then the sum of the costs of optimizing the individual boxes would be greater than or equal to the cost of optimizing all the boxes together.

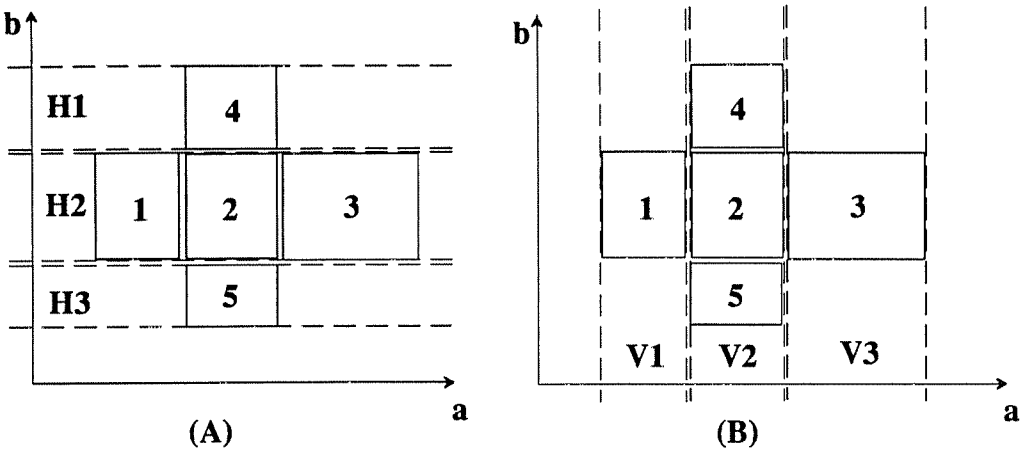
Merging adjacent boxes in Figure 4.2 and optimizing, we would choose the better of the following two plans:

Plan 1: Optimize rectangles (1, 2, 3), 4, and 5
Plan 2: Optimize rectangles (4, 2, 5), 1, and 3.

The disadvantage of this method is that it may result in more pages being fetched from disk than are really necessary. Consider Plan 1, for example. Assume that rectangle 4 was optimized using the index on the **a** attribute and that rectangle 5 was optimized using the index on the **b** attribute. Clearly the tuples in the solution space corresponding to rectangle 5 would be present in the relation pages fetched when optimizing rectangle 4 using the index on attribute **a**. This is because both the rectangles lie in the same range of attribute **a** values. In fact, when box 4 is optimized using the index on **a**, tuples in the entire shaded region in Figure 4.2 can be accessed. This observation leads us to our next method.

### 4.4.3. Covering By Subspaces

This method seeks to cover the solution space by subspaces. Assuming single-attribute indices, these subspaces are bounded above and below along one attribute (dimension) and unbounded along all the other attributes (dimensions). The width of a subspace, along the bounded attribute, is equal to the range of the attribute values that tuples accessed via the index on that attribute will have. As shown in Figure 4.4, the entire solution space can be covered by subspaces in three different ways. Notice that each subspace has a



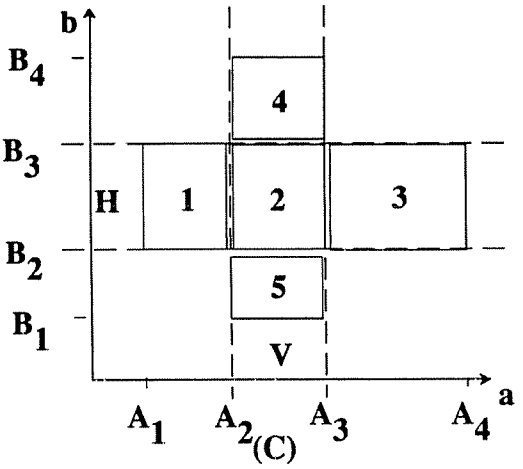




**Figure 4.4**

cost associated with it which is equal to the number of pages fetched via the index on the attribute on which the subspace is bounded. Assume that the least expensive way of covering the solution space in Figure 4.4 was by covering it with one horizontal subspace and one vertical subspace (Figure 4.4 (C)). In other words, we have

COST (H) + COST (V) < COST (H1) + COST (H2) + COST (H3)

and

COST (H) + COST (V) < COST (V1) + COST (V2) + COST (V3)

Since H and V overlap, duplicates are possible in the result if we are not careful. Assume we first use the index on **b** and retrieve tuples covered by the H subspace. Since H covers rectangles 1, 2, and 3, we will retrieve all tuples that already satisfy the qualification

$(\mathbf{s.b} \geq B_2$ AND $\mathbf{s.b} \leq B_3)$.

However, we are interested only in tuples that also satisfy

$(\mathbf{s.a} \geq A_1$ AND $\mathbf{s.a} \leq A_4)$.

Now, when we retrieve tuples covered by subspace V, using the index on **a**, we will get tuples that automatically satisfy

$(\mathbf{s.a} \geq A_2$ AND $\mathbf{s.a} \leq A_3)$.

In order to avoid duplicates, we must ignore tuples in rectangle 2 and keep those in rectangles 4 and 5. Hence, when retrieving tuples via the index on **a**, only those tuples that satisfy

$(\mathbf{s.b} \geq B_1$ AND $\mathbf{s.b} < B_2)$ or $(\mathbf{s.b} > B_3$ AND $\mathbf{s.b} \leq B_4)$

should be added to the result relation.

In Section 4.6, we will show that the problem of finding subspaces to cover the solution space such that the sum of their costs is minimized is equivalent to finding a minimum cost vertex cover in a hypergraph. We will be interested in a restricted class of hypergraphs that we will define in the next section.

### 4.5. Minimum Cost Vertex Cover in a Hypergraph

We define the restricted class of hypergraphs as follows:

A restricted hypergraph is a hypergraph H = (V, E) where

$V = V_1 \cup V_2 \cup \ ... \ \cup V_m$ such that $V_i \cap V_j = \emptyset$, $1 \leq i, j \leq m$
and
$E \subseteq V_1 \times V_2 \times \ ... \ \times V_m$.

In other words, the restricted hypergraph[15] consists of disjoint partitions of vertices such that each hyperedge passes through a single vertex in each partition. Figure 4.5 shows an example of a hypergraph with three vertex partitions ($V_1$, $V_2$ and $V_3$) and five hyperedges ($e_1$ through $e_5$). The problem of finding a minimum vertex cover in a hypergraph is the following:

[15]Henceforth, we will only deal with this restricted class of hypergraphs.

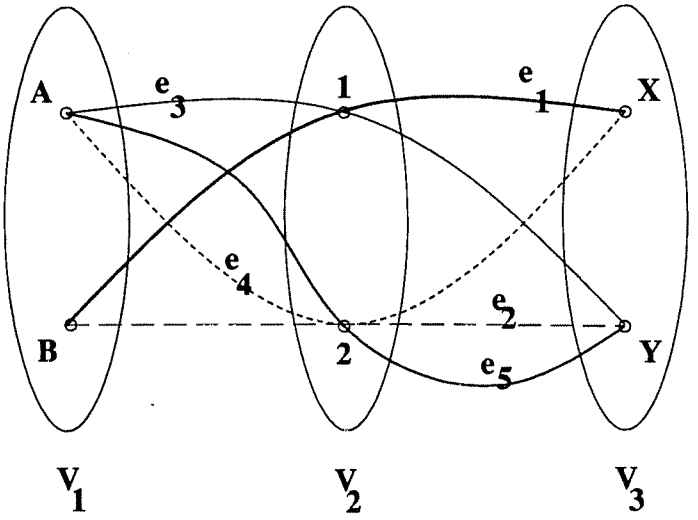



**Figure 4.5**

Given a positive cost for each vertex in V, find a subset of vertices in V such that the sum of their costs is minimum and every hyperedge in E is incident on at least one vertex in this subset.

Formally, the problem is to find $S \subseteq V$ that minimizes

$$\sum_{\omega \in S} \text{cost}(\omega)$$

such that every hyperedge $e_i$, $1 \leq i \leq |E|$, has at least one vertex in common with S.

A subset of vertices that covers every hyperedge is called a **vertex cover** and the subset with the minimum cost that covers every hyperedge is called the **minimum cost vertex cover**.

We will now show that covering the solution space of a query by subspaces such that their cost is minimized can be formulated as a problem of finding a minimum vertex cover in a hypergraph. In fact, the two problems are equivalent. In other words, given one of the problems we can uniquely construct the other.

### 4.6. Equivalence of the Two Problems

Consider the example of the two disjunct query of Section 4.2 again.

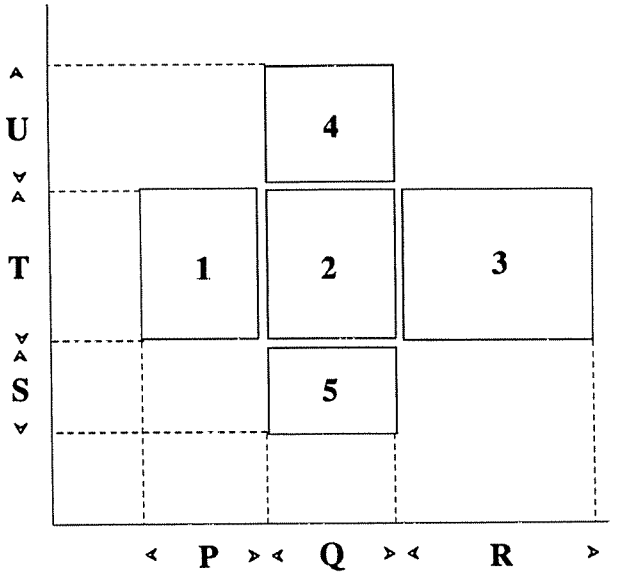
retrieve (TL)
where
(**s.a** $\geq A_1$ AND **s.a** $\leq A_4$ AND **s.b** $\geq B_2$ AND **s.b** $\leq B_3$) --- ($D_1$)
OR

$(\mathbf{s.a} \geq A_2$ AND $\mathbf{s.a} \leq A_3$ AND $\mathbf{s.b} \geq B_1$ AND $\mathbf{s.b} \leq B_4)$ --- $(D_2)$

The solution space of this two disjunct query is shown in Figure 4.6. By projecting the corner points of each of the rectangles corresponding to the two disjuncts on to the coordinate axes, we get the intervals P, Q, and R on the horizontal axis and the intervals S, T, and U on the vertical axis. If there are d disjuncts (boxes) in the query, the number of intervals on each axis is at most 2d -1. We can associate a subspace uniquely with each of these intervals. For example, we can associate a vertical subspace with Q that covers boxes (rectangles) 4, 2, and 5. Similarly, we can associate a horizontal subspace with S that covers box 5. Each of the disjoint boxes (1 through 5) can be obtained uniquely by the intersection of a horizontal subspace and a vertical subspace. For example, box 2 is obtained by the intersection of the two subspaces associated with intervals Q and T respectively. In the m-dimensional case, each box will be obtained by the intersection of m subspaces, each from a different dimension. As already seen in Section 4.4.3, we can associate a cost with each of the subspaces.

We now proceed to show how the equivalent hypergraph, also shown in Figure 4.6, is constructed. Associate a vertex with each subspace (interval). The intervals on each axis correspond to vertices in one vertex partition of the hypergraph. This results in the two disjoint sets of vertices {P', Q', R'} and {S', T', U'}. The cost of each vertex is the same as the cost of the corresponding subspace. A hyperedge exists between two vertices, each vertex belonging to a different vertex partition, if and only if the region produced by the intersection of the subspaces corresponding to the two vertices is in the solution space[16]. For example, subspaces Q and T intersect to give box 2. The hyperedge that corresponds to this box is the edge that passes through Q' and T' in the hypergraph. It is now easy to see that this hypergraph has three minimal vertex covers, viz., {P', Q', R'} which corresponds to covering the solution space with only vertical subspaces (as shown in Figure 4.4), {S', T', U'} which corresponds to covering the solution space with only horizontal subspaces, and finally, {Q', T'} which corresponds to covering the solution space with a horizontal subspace and a vertical subspace. The hypergraph in Figure 4.6 is a bipartite graph, as there are only two vertex partitions. Clearly, given the hypergraph, we can uniquely construct the solution space. Thus, the two problems are equivalent.

[16]This same criterion for the existence of a hyperedge extends to more than two dimensions. The complexity of the brute force algorithm to test if a box is in the solution space is $O((2d - 1)^m)$ as there are at most $(2d - 1)^m$ boxes.



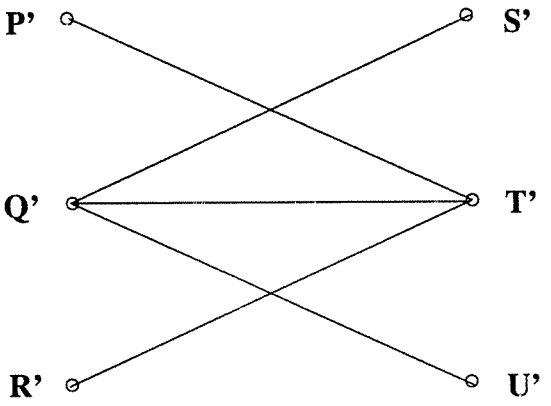




**Figure 4.6**

### 4.7. Why Formulate as a Minimum Vertex Cover Problem?

In the previous section we demonstrated that the problem of covering the solution space with subspaces of minimum cost is equivalent to finding a minimum cost vertex cover in a hypergraph. We can prove some important results about the minimum vertex cover problem using well known results from different fields such as graph theory, linear programming, and integer programming. These results then apply to the former problem as the two are equivalent. We will also show how a minimum vertex cover can be

found in an arbitrary hypergraph.

Interestingly, the minimum vertex cover problem can also be formulated as an Integer Programming (IP) problem as follows. Let **A** represent the incidence matrix (|E| x |V|) of a hypergraph.

$$A[i, j] = \begin{cases} 0 & \text{if the ith hyperedge is not incident on the jth vertex.} \\ 1 & \text{if the ith hyperedge is incident on the jth vertex.} \end{cases}$$

Let **x** be a vector of size |V| and **C** be the positive cost vector associated with the vertices. Then, the associated IP is

minimize **Cx**
subject to $\mathbf{Ax} \geq \mathbf{1}$
where $\mathbf{x_j}$ are 0-1 variables and **1** is a vector of 1's.

The condition $\mathbf{Ax} \geq \mathbf{1}$ specifies that every hyperedge must be covered.

In the final solution,

$$\mathbf{x_j} = \begin{cases} 0 & \text{if the jth vertex is not in minimum vertex cover.} \\ 1 & \text{if the jth vertex is in minimum vertex cover.} \end{cases}$$

We will now show how optimal access plans can be generated efficiently for a multi-disjunct query on a relation with at most two relevant indices.

### 4.8. A Query With at Most Two Relevant Indexes

A query involving n attributes on a relation with t attributes will be characterized as an m-dimensional query, $m \leq n \leq t$, if m of the n attributes have indices built on them. In this section we will focus our attention on optimizing 1-dimensional and 2-dimensional queries. We will show in the next section that solving m-dimensional queries, $m \geq 3$, is a harder problem.

Consider a relation s with attributes $\mathbf{a_1}$, $\mathbf{a_2}$, ... , $\mathbf{a_t}$. Assume that attributes $\mathbf{a_1}$ and $\mathbf{a_2}$ have indices built on them. The hypergraph corresponding to the solution space of a 2-dimensional query on s is constructed, as described in Section 4.6. We know that the cost (in pages) of the vertices in the partitions associated with the non-indexed attributes will be equal to the number of pages in the relation. Hence, these vertices may be removed from the hypergraph because we do not want them to participate in the minimum cost vertex cover.

The incidence matrix of a bipartite graph is *totally unimodular* [Bazaraa77]. Hence, all basic solutions will automatically be integral. Therefore, the IP associated with a 2-dimensional query can be solved as a Linear Program (LP) [Bazaraa77]. This is important because it is much harder to solve an IP than it is to solve an LP. Unfortunately, the incidence matrix of a hypergraph with more than two vertex partitions is not totally unimodular. Consider, for example, the hypergraph shown in Figure 4.5. It can be shown that its incidence matrix has a rank of 4. The 4x4 square sub-matrix formed by the rows corresponding to edges (e1, e2, e3 and e4) and by the columns corresponding to vertices (A, 2, Y, X) has a determinant of 2. Hence 3 or more dimensional queries cannot be solved as an LP. In fact, we will show in the next section that finding a minimum cost vertex cover in a hypergraph with 3 or more vertex partitions is NP-complete.

### 4.9. Optimizing m-Dimensional Queries ($m \geq 3$)

We will first prove that the problem P, of finding a minimum cost vertex cover for hypergraphs of m-dimensional queries ($m \geq 3$), is NP-complete. We state P formally:

**Instance**: Given a k-dimensional ($k \geq 3$) hypergraph $H = (V', E')$, cost $C(v') > 0$, $v' \in V'$, and positive integer $\Omega \leq \sum_{v' \in V'} C(v')$.

**Question**: Does there exist a subset S of V' such that S is a vertex cover and $\sum_{v' \in S} C(v') \leq \Omega$?

**Theorem**: Problem P is NP-complete.

**Proof**: Clearly P is in NP as a nondeterministic algorithm can pick a subset of V whose cost is $\omega \leq \Omega$ and check in polynomial time if the subset is a vertex cover.

We shall make use of the following two results in the rest of our proof.

Finding a minimum weighted vertex cover in a cubic planar graph is NP-complete [Garey76] --- (1).

A graph of degree k is k-colorable[17] [Lovasz79] --- (2).

It can be shown, using (1), that

finding a minimum weighted vertex cover for planar graphs of degree k ($k \geq 3$) is NP-complete --- (3)

---

[17]We are assuming that the graph is not complete. If the graph is complete, we will need k+1 colors.

We shall reduce (3) to the problem of finding a minimum vertex cover in a hypergraph as follows:

Consider a planar graph G = (V, E) with degree k ($k \geq 3$). Each vertex $\in$ V has a positive cost given by C(v). G can be colored with k colors in polynomial time[18]. This partitions the vertex set into k partitions or independent sets $V_1, V_2, ..., V_k$.

We now construct the hypergraph, H = (V', E'), as follows:

The vertices of H, $V' = \{v'_i \mid v_i \in V\} \cup \{p_i : 1 \leq i \leq k\}$

$C(v'_i) = C(v_i)$ and $C(p_i) = \infty$

The hyperedges of H, $E' = \{e' \mid e \in E\}$

For every edge, $e \in E$, $e = \{v_i, v_j\}$, $i < j$, $v_i \in V_i$, $v_j \in V_j$, construct a hyperedge

$e' = \{p_1, p_2, \cdots, p_{i-1}, v'_i, p_{i+1}, \cdots, p_{j-1}, v'_j, p_{j+1}, \cdots, p_k\}$

We now claim that a weighted vertex cover of weight $\omega \leq \Omega$ exists in G if and only if a weighted vertex cover of weight $\omega \leq \Omega$ exists in H. If S is a vertex cover of weight $\omega \leq \Omega$ in G, then $S' = \{ v' \mid v \in S\}$ is clearly a vertex cover of weight $\omega \leq \Omega$ in H. If S' is a vertex cover of weight $\omega \leq \Omega$ in H, then S' cannot contain any of the $p_i$'s, since otherwise, its weight would exceed $\Omega$. Also $S = \{ v \mid v' \in S'\}$ is a vertex cover of G and is of weight $\omega \leq \Omega$.

Therefore, it follows that the problem of finding a minimum cost vertex cover in a hypergraph is NP-complete. □

This raises the question: How does one optimize m-dimensional ($m \geq 3$) queries? One obvious but not very efficient method is to solve the associated IP. This will, however, give the optimal access paths for the query. If the query is to be run a large number of times it may well be worth solving the IP. If the number of vertices is small, it may also be feasible to find the minimum vertex cover by enumerating all possible vertex covers. This brute force algorithm will be of complexity $O(2^{|V|})$.

There is a rather simple way of determining if we should employ the brute force technique. Let us assume we have an m-dimensional query with **d** disjuncts. Therefore, the number of vertices, |V|, in our hypergraph is at most m * (2d -1). Let the vector **x** be represented as a bit vector of size |V|. We can store the edges (rows of the incidence matrix) as |E| bit vectors. Thus, to find if a vector $\mathbf{x}^1$ is a vertex cover, we need to perform |E| bitwise AND operations. To find all possible vertex covers, it should take us

[18]For graphs of degree k, $k \geq 3$, see [Lovasz79] pp. 354-355 for the coloring algorithm.

C * Time for |E| bitwise AND operations * $2^{|V|}$ time units.

where C is some positive constant whose value can be determined experimentally. If this is greater than the time for scanning the entire relation, we know that it is not beneficial to find a minimum cost vertex cover for *ad hoc* queries. Instead, we perform a file scan of the entire relation. The brute force technique may not be practical in an environment where users present *ad hoc* queries. In such an environment, it may be desirable to obtain an approximate solution that is *not too far* from the optimal, provided that the approximate solution can be generated efficiently. For an m-dimensional query there is an O(|E|) algorithm that guarantees a solution that is at most $W_{opt}$*m where $W_{opt}$ is the optimal cost.

### 4.10. The O(|E|) Approximation Algorithm

An approximation algorithm for the weighted vertex cover problem for a general graph was presented in [Bar-Yehuda81]. The running time of the algorithm was linear in |E|. We present their algorithm here. The input is an m-dimensional hypergraph, with non-negative vertex costs, denoted by $\omega_i$ for the ith vertex. SW(i) denotes the residual cost of the ith vertex. **I** denotes the set of indices of vertices already in the cover; **J** denotes the set of hyperedges that have not yet been covered. F(j) denotes the set of vertices incident on the jth hyperedge while S(i) denotes the set of edges incident on the ith vertex.

SW(i) = $\omega_i$ for all i, $1 \le i \le |V|$.
**I** = ∅, **J** = E.
**while J** ≠ ∅ **do**
    Let j ∈ **J**.
    M = Minimum { SW(i)|i ∈ F(j)}
    Let k ∈ F(j) such that SW(k) = M.
    For all i ∈ F(j) SW(i) = SW(i) - M.
    **I** = **I** ∪ {k}, **J** = **J** - S(k).
**end_while**
**halt**

It is shown in [Bar-Yehuda81] that the total cost, W, of the vertex cover **I** produced by the above algorithm, satisfies

$$W \le W_{opt} * \text{Max } |F(j) \cap \mathbf{I}|, \ 1 \le j \le |E|.$$

We know that, by definition, |F(j)| = m, for every hyperedge. This implies that Max $|F(j) \cap \mathbf{I}| \le m$. Therefore $W \le W_{opt} * m$.

### 4.11. Optimizing with Multi-Dimensional Indices

So far we have discussed optimization in the context of single-attribute indices only. Several multi-dimensional index structures for accessing point data have been proposed in the literature. The two well known multi-dimensional index structures for point data are the KDB-tree [Robinson81] and the Grid file [Nievergelt84]. Multi-dimensional index structures find wide applications in the areas of geographical databases, image databases, VLSI databases, etc.

As discussed earlier in Section 4.4.3, optimizing a multi-dimensional query using single-dimensional indices involved the covering of the solution space by subspaces that were bounded along only one dimension. However, as seen in Figure 4.2, this results in fetching tuples (pages) that are not in the solution space. Let the set of attributes in the query be $A_Q$ and the set of attributes on which an index is built be $A_I$. Optimizing a multi-dimensional query with a multi-dimensional index $I_i$ is equivalent to covering the solution space with subspaces that are bounded along $|A_Q \cap A_{I_i}|$ dimensions. The question that arises here is the following: How does one extend the hypergraph model to incorporate optimization with multi-dimensional indices? It turns out that the extension is simple and elegant.

The covering of part of the solution space by a subspace bounded along one dimension was equivalent to covering a hyperedge with a single vertex in the corresponding hypergraph. Similarly, covering part of the solution space with a subspace bounded along k dimensions is equivalent to covering a hyperedge with k vertices. When a hyperedge is covered with k ($k \geq 1$) vertices, all hyperedges that are incident on these k vertices are also covered. The cost of covering hyperedges with k vertices is equal to the number of pages that will be retrieved in order to fetch tuples that lie in the region of intersection of the k subspaces (bounded along one dimension) corresponding to the k vertices. For example, if we had a two dimensional index for the query whose hypergraph is shown in Figure 4.6, we would have optimally covered each edge of the hypergraph with its two end vertices.

Instead of designing an algorithm where a hyperedge could be covered by a set of vertices rather than by a single vertex, we chose to modify the original hypergraph so that every edge in the modified hypergraph can be covered with a single vertex. The advantage lies in the fact that we can then use existing vertex cover algorithms. Let the original hypergraph be $H_o = (V, E)$ where

$$V = V_1 \cup V_2 \cup \ldots \cup V_n \text{ such that } V_i \cap V_j = \emptyset,\ 1 \leq i, j \leq n$$

and
$E \subseteq V_1 \times V_2 \times \ldots \times V_n$.

The set of attributes in the query is $A_Q = \{1, 2, ..., n\}$. Let $S_I$ denote the set of relevant indices. Assuming that there are m indices, $S_I$ is given by $\{1, 2, ..., m\}$. Let $A_i$ denote the set of attributes on which the ith index is built. Let $B = A_Q - \bigcup_{i \in S_I} A_i$. We define the modified hypergraph H' = (W, E') as follows:

$W = W_1 \cup W_2 \cup \ldots \cup W_p$ such that $W_i \cap W_j = \emptyset$, $1 \le i, j \le n$
and
$E' \subseteq W_1 \times W_2 \times \ldots \times W_p$.

where $p = |S_I| + |B| = m + |B|$.

The $W_i$'s are given by[19]:

$$W_i = \prod_{j \in A_i} V_j, \quad 1 \le i \le m.$$

$$W_i = V_j, \quad j \in B, \quad m+1 \le i \le p.$$

There is a 1-1 correspondence between the edges in E and E'. For every edge

$e \in E = \langle v_1, v_2, \ldots, v_n \rangle$, $v_i \in V_i$, $1 \le i \le n$,

the corresponding edge $e' \in E'$ is given by

$\langle w_1, w_2, ..., w_p \rangle$, $w_i \in W_i$, $1 \le i \le p$,

where the $w_i$'s are given by:

$$w_i = \prod_{j \in A_i} v_j, \quad 1 \le i \le m.$$

$$w_i = v_j, \quad j \in B, \quad m+1 \le i \le p.$$

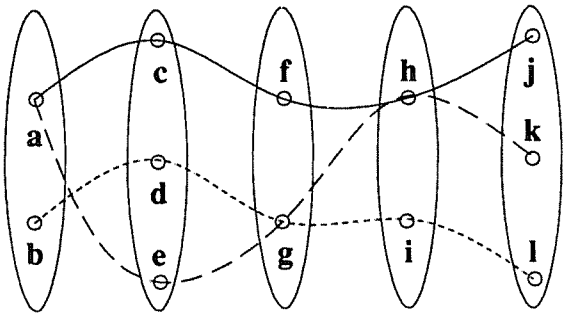
The cost of each vertex in W is calculated as described above.

We will illustrate the above transformation with an example. Figure 4.7(A) shows a 5-dimensional hypergraph ($H_0$) with three hyperedges. $A_Q = \{1, 2, 3, 4, 5\}$. Assume two indices ($S_I = \{1, 2\}$) such that $A_1 = \{1, 2\}$ and $A_2 = \{2, 3, 4\}$. Therefore $B = \{5\}$ and $p = |S_I| + |B| = 2 + 1 = 3$. Figure 4.7(B) shows the modified hypergraph (H') with 3 vertex partitions. The first vertex partition in H' was obtained by taking the cross product of the first and second vertex partitions in $H_0$. Similarly, the second vertex partition in H' was obtained by taking the cartesian product of the second, third, and fourth vertex partitions in $H_0$. The

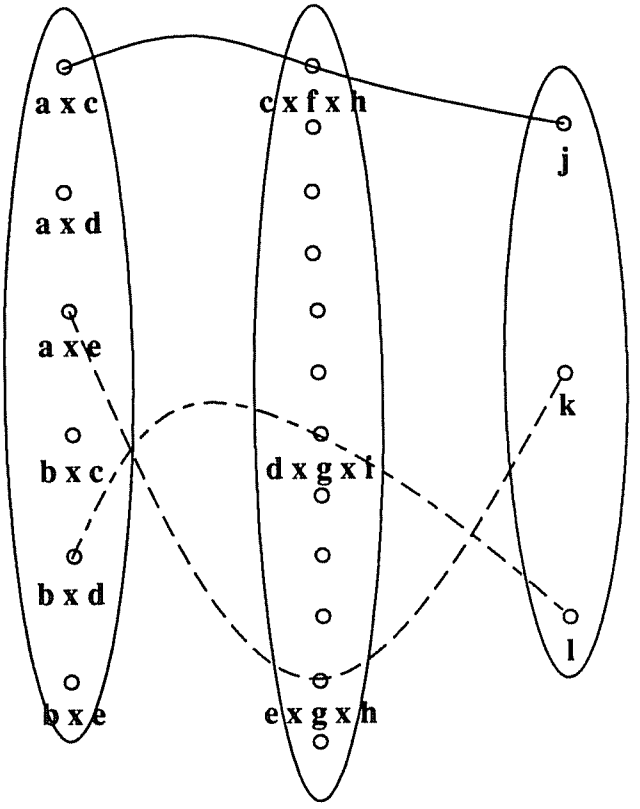
---

[19] $\prod_{j \in A_i} V_j$ is equivalent to $V_{i_1} \times V_{i_2} \times \ldots \times V_{i_k}$, $i_k \in A_i$.

third vertex partition is identical to that of the fifth vertex partition of $H_0$. The edge $(a \cup c \cup f \cup h \cup j)$ in $H_0$ gives rise to the edge $((a \times c) \cup (c \times f \times h) \cup j))$ in H'. The other two edges are drawn similarly.




(A)




(B)

**Figure 4.7**

# CHAPTER 5

## A NEW APPROXIMATION ALGORITHM FOR THE VERTEX COVER PROBLEM

The general problem of finding a minimum cost vertex cover in graphs in NP-complete [Garey79]. In Chapter 4 we saw that the problem of finding a minimum cost vertex cover problem in hypergraphs is NP-complete. This has necessarily led to the development of approximation algorithms and heuristics for solving the vertex cover problem. The most popular is the Bar-Yehuda [Bar-Yehuda81] approximation algorithm presented in Section 4.10. The [Bar-Yehuda81] paper also contains a brief survey of other vertex cover algorithms and the relevant references. The Bar-Yehuda algorithm considers only one edge at every iteration. The cheapest vertex on that edge is included in the vertex cover. Thus, the information used at every iteration by the Bar-Yehuda algorithm is very localized. Intuitively, we felt that using more information such as the costs and degrees of the neighbors of a vertex can, on the average, lead to a better vertex cover algorithm. The Bar-Yehuda algorithm is very appealing because its authors were able to derive an upper bound on the performance of the algorithm. Deriving an analytical bound on the performance of a heuristic is generally difficult.

In this chapter we present a new approximation algorithm for simple graphs along with the motivation behind it. We will then extend the algorithm to hypergraphs. Unfortunately, we were not able to derive an analytical bound on the performance of our algorithm. In order to demonstrate the quality of our algorithm, we generated random graphs in the class $G_{n, p}$ [Bollobas85]. For graphs in the class $G_{n, p}$ with unit cost vertices, the expected value of the minimum cost of the vertex cover can be very accurately computed. We will show that our algorithm gives vertex cover costs that are much closer to the expected value than those obtained by the Bar-Yehuda algorithm. We will also present the results of other experiments on random hypergraphs that demonstrate empirically that our heuristic performs much better, on the average, than the Bar-Yehuda algorithm on random hypergraphs.

Why did we choose to test the efficacy of our algorithms on random hypergraphs rather than on hypergraphs that correspond to typical database queries? The answer is that we do not know what typical single-relation, multi-disjunct queries are. There are no benchmarks for multiple-disjunct queries. The Wisconsin Benchmark [Bitton83] was designed only for single-disjunct queries. In addition, random hypergraphs lend themselves to easy analytical analysis and thus are at least a good starting point.

### 5.1. The Approximation Algorithm for Simple Graphs

The algorithm consists of two parts. The first part ($H_1$) follows from a simple observation. Let $v_1$, $v_2$, ..., $v_n$ be all the vertices of degree 1 that are connected to a common vertex v in graph G as seen in Figure 5.1. The edges $v_1$--v, $v_2$--v, ..., $v_n$--v can be covered either by the singleton set {v} or by the set {$v_1$, $v_2$, ..., $v_n$}. Clearly, if $\sum_{i=1}^{n} cost(v_i) \geq cost(v)$, then v will be part of the minimum cost vertex cover. Therefore we must choose v in the cover. The point here is that, whenever possible, we should exclude unit degree vertices from the vertex cover. If v is chosen in the cover, we remove v and all the edges incident on it from the graph. In an acyclic graph with unit cost vertices, $H_1$ can be repeatedly applied to obtain an optimal solution.

On the other hand, if $\sum_{i=1}^{n} cost(v_i) < cost(v)$, and there are no other edges in the same component,

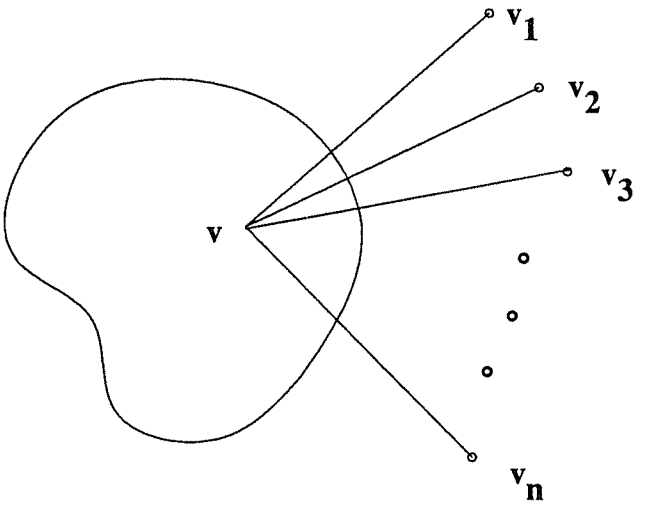



**Figure 5.1**

the set $\{v_1, v_2, ..., v_n\}$ will be the optimal vertex cover. We will return to this point at the end of the section.

The motivation for the second part ($H_2$), which is a heuristic, came partly from $H_1$. We apply $H_2$ only if $H_1$ cannot be applied. Let N(v) denote the set of neighbors of vertex v. We calculate Q(v) for every vertex v in G, where Q(v) is given by:

$$Q(v) = \sum_{v_i \in N(v)} \frac{cost(v_i)}{degree(v_i)} - cost(v)$$

The vertex with the maximum Q value is included in the cover. Ties may be broken arbitrarily. Once a vertex is chosen in the cover, we remove that vertex and all the edges incident on it from the graph. We now present our approximation algorithm in pseudo-code.

**Algorithm**

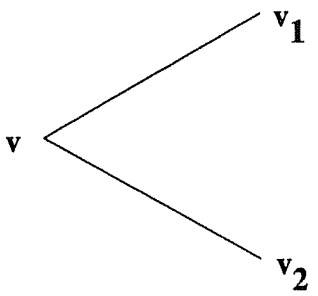
```
while (edges exist in the graph)
begin
    while (H1 can be applied)
    begin
        Apply H1.
        Remove the corresponding vertex and the edges incident on it.
    end while

    if (edges exist in the graph)
    begin
        Calculate Q(v) for every vertex in the graph.
        Include v with maximum Q(v) in the cover.
        Remove v and the edges incident on it.
    end if
end while
```

**End Algorithm**

We must point out that it is important to apply $H_2$ only if $H_1$ cannot be applied. Let us illustrate this with an example.

**Example**: Consider the graph shown in Figure 5.2. Let $cost(v) = cost(v_1) = 5$, and $cost(v_2) = 1$. Since $cost(v_1) + cost(v_2) > cost(v)$, $H_1$ can be applied and the optimal vertex cover is the set $\{v\}$. However, if $H_2$ is applied, we will get the following Q values:

$$Q(v) = \frac{cost(v_1)}{1} + \frac{cost(v_2)}{1} - cost(v) = 5 + 1 - 5 = 1.$$




$$\mathbf{cost(v) = cost(v_1) = 5,\ cost(v_2) = 1.}$$

$$\mathbf{Q(v) = 1,\ Q(v_1) = -2.5,\ Q(v_2) = 1.5.}$$

**Figure 5.2**

$$Q(v_1) = \frac{cost(v)}{2} - cost(v_1) = \frac{5}{2} - 5 = -2.5.$$

$$Q(v_2) = \frac{cost(v)}{2} - cost(v_2) = \frac{5}{2} - 1 = 1.5.$$

We would thus choose $v_2$ to be in the vertex cover, leading to a non-optimal solution in this case. In general, applying $H_2$ before applying $H_1$ can lead to vertex covers of higher cost.

Let us reconsider the graph in Figure 5.1. Assume that $\sum_{i=1}^{n} cost(v_i) < cost(v)$, and that there are no other edges in the same component. We know then that the set $\{v_1, v_2, ..., v_n\}$ will be the optimal vertex cover. What would happen if we apply $H_2$ here before we apply $H_1$?

**Lemma:** Under these circumstances $H_2$ will not pick vertex v.

**Proof:** We have

$$cost(v) > \sum_{i=1}^{n} cost(v_i) \tag{1}$$

Without loss of generality, let

$$cost(v_1) \leq cost(v_i),\ 1 \leq i \leq n \tag{2}$$

Therefore

$$cost(v_1) \leq \frac{\sum_{i=1}^{n} cost(v_i)}{n} \tag{3}$$

The Q values are as follows:

$$Q(v) = \sum_{i=1}^{n} cost(v_i) - cost(v) \tag{4}$$

and

$$Q(v_i) = \frac{cost(v)}{n} - cost(v_i),\ 1 \le i \le n \tag{5}$$

From (2) it follows that $Q(v_1) \ge Q(v_i)$, $1 \le i \le n$. $H_2$ will pick vertex v if and only if $Q(v) \ge Q(v_1)$. Assuming $Q(v) \ge Q(v_1)$, we get from (4) and (5)

$$\sum_{i=1}^{n} cost(v_i) - cost(v) \ge \frac{cost(v)}{n} - cost(v_1)$$

Rearranging, we get

$$\sum_{i=1}^{n} cost(v_i) + cost(v_1) \ge cost(v)(1+\frac{1}{n}) \tag{6}$$

From (1) we get

$$cost(v)(1+\frac{1}{n}) > \sum_{i=1}^{n} cost(v_i)(1+\frac{1}{n}) \tag{7}$$

By transitivity and from (6) and (7), we have

$$\sum_{i=1}^{n} cost(v_i) + cost(v_1) > \sum_{i=1}^{n} cost(v_i)(1+\frac{1}{n})$$

Simplifying, we get that

$$cost(v_1) > \frac{\sum_{i=1}^{n} cost(v_i)}{n}$$

This statement contradicts (3). Therefore, under these circumstances, $H_2$ will choose the optimal vertex cover. □

The reason that we have presented the above lemma is that a similar lemma for hypergraphs turns out not to be true. We will show this in Section 5.5 when we discuss the extension of the algorithm to hypergraphs.

What is the worst case complexity of the algorithm? Let us assume that the graph is stored in the form of an adjacency matrix. To determine if $H_1$ is applicable, we need to look at the neighbors of a vertex. Finding the neighbors of a vertex using an adjacency matrix takes O(n) steps, where n is the number of vertices in the graph. Therefore, we require $O(n^2)$ to find the neighbors of all n vertices.

Before applying $H_2$ we need to know the neighbors of a vertex (O(n) steps) and their degrees. We can precompute the degrees of all the vertices in $O(n^2)$ steps. Thus, we can find the Q value of one vertex in O(n) steps. To find the Q value of every vertex we need $O(n^2)$ steps.

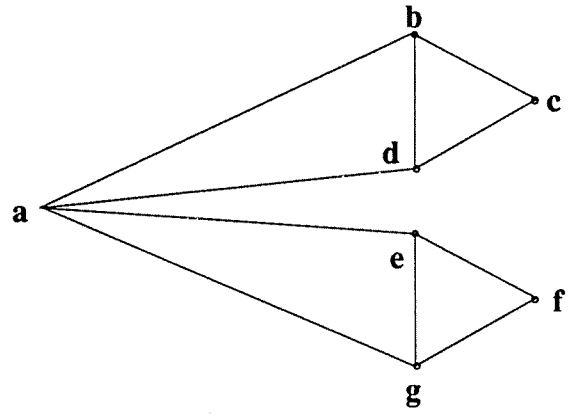
In every pass through the body of the outer while loop of the algorithm, we will remove at least one vertex. If the graph is a clique, we will need to remove (n - 1) vertices before all the edges are covered. Therefore, the worst case running time of our algorithm is $O(n^3)$ which is equivalent to O(n * e), where e is the total number of edges in the graph. On the other hand, the running time of the Bar-Yehuda Algorithm for graphs is O(e).

### 5.2. A Pathological Example

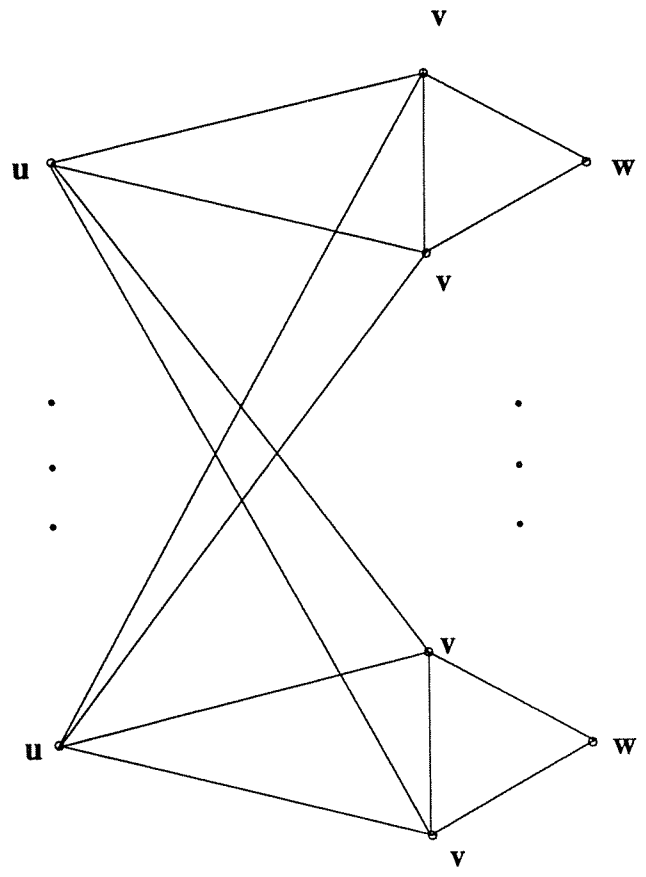
Not having been able to come up with an analytical bound on the performance of our approximation algorithm, we were nevertheless interested in constructing a pathological example where the ratio of the solution yielded by our approximation algorithm ($W_a$) to that of the optimal solution ($W_{opt}$) would be as large as possible. We already know that for simple graphs $\frac{W_b}{W_{opt}} \leq 2$, where $W_b$ is the solution obtained by the Bar-Yehuda Algorithm. We should point out that the example we present in this section is pathological only in the sense that we have not yet discovered a graph with a higher $\frac{W_a}{W_{opt}}$ ratio.

The smallest graph[20] (in terms of the number of vertices) for which our approximation algorithm gives a non optimal vertex cover is shown in Figure 5.3(A). Assuming unit cost vertices, the optimal vertex cover is the set {b, d, e, g}. Vertex a has the largest Q value. Therefore, our algorithm first picks vertex a leaving the two triangles bcd and efg. This leads to a cover that consists of five vertices rather than four vertices. Thus,. $\frac{W_a}{W_{opt}} = 1.25$. We generalized the graph in Figure 5.3(A) to the one shown in Figure 5.3(B) such that it contains m > 0 vertices on the left and n > 0 triangles on the right. Every vertex on the left is connected to two vertices of every triangle and vice versa. The third vertex of each triangle on the extreme right is connected only to the other two vertices of the corresponding triangle. We will now

[20]Graphs on six vertices can have a maximum of 15 edges. For all the $2^{15}$ graphs on six vertices (with unit costs), our algorithm gives the optimal solution.




(A)




(B)

**Figure 5.3**

analytically show that $\frac{W_a}{W_{opt}}$ can asymptotically reach 2.

Let u, v, and w be representative vertices as shown. A vertex on the extreme left in Figure 5.3(B) is a u vertex, a w vertex is a vertex on the extreme right, and all the remaining vertices are v vertices. degree(u) = 2n, degree(v) = m + 2, and degree(w) = 2. Let cost(u) = $C_1 > 0$, and cost(v) = cost(w) = $C_2 > 0$. Clearly, at least two vertices of every triangle must be included in any vertex cover. Therefore, the optimal vertex cover is the set of v vertices, and the cost of the optimal vertex cover is $2nC_2$. Computing the Q values for each of the three representative vertices we get,

$$Q(u, m, n) = 2n\frac{C_2}{m+2} - C_1,$$

$$Q(v, m, n) = m\frac{C_1}{2n} + \frac{C_2}{m+2} + \frac{C_2}{2} - C_2,$$

and

$$Q(w, m, n) = 2\frac{C_2}{m+2} - C_2.$$

We will first derive a few results.

For all positive values of m and n, we have:

$$m\frac{C_1}{2n} + \frac{C_2}{2} > \frac{C_2}{m+2}.$$

Adding $\frac{C_2}{m+2} - C_2$ to both sides, we get

$$Q(v, m, n) > Q(w, m, n) \tag{1}$$

for all positive values of m and n. Therefore, a w vertex will never be picked by our algorithm.

It is easy to see that

$$Q(v, m, n-1) > Q(v, m, n) \text{ and } Q(u, m, n) > Q(u, m, n-1). \tag{2}$$

Therefore,

$$\text{If } Q(v, m, n) \geq Q(u, m, n), \text{ then it follows that } Q(v, m, n-1) > Q(u,m, n-1). \tag{3}$$

If $Q(u, m, n) \geq Q(v, m, n)$, we have

$$\frac{(2n-1)C_2}{(m+2)} - C_1 \geq \frac{mC_1}{2n} - \frac{C_2}{2.}$$

This implies that

$$(2n-1)\frac{C_2}{(m+1)} - C_1 > (m-1)\frac{C_1}{2n} - \frac{C_2}{2.}$$

Adding $\frac{C_2}{m+1}$ to both sides, we get

$$Q(u, m-1, n) = 2n\frac{C_2}{(m+1)} - C_1 > (m-1)\frac{C_1}{2n} + \frac{C_2}{m+1} - \frac{C_2}{2} = Q(v, m-1, n).$$

Therefore,

$$\text{If } Q(u, m, n) \geq Q(v, m, n), \text{ it follows that } Q(u, m-1, n) > Q(v, m-1, n). \tag{4}$$

We shall now consider three cases:

**Case 1**: $Q(v, m, n) > Q(u, m, n)$.

A v vertex in some triangle will be chosen by the algorithm. After the edges incident on v are removed from the graph, the w vertex belonging to that triangle will have a degree of 1. This will cause the other v vertex of the triangle also to be included in the vertex cover (by $H_1$). This will result in a graph with m points on the left side and (n-1) triangles on the right. It follows from (3) that v vertices will always be chosen in subsequent iterations. Therefore, our algorithm will yield the optimal solution.

**Case 2**: $Q(v, m, n) < Q(u, m, n)$.

This will cause a u vertex to be chosen. In fact, it follows from (4) that a u vertex will be chosen on the first m iterations. This will leave the n triangles in the graph. Every triangle must be covered by two vertices each. Thus, the cost of the vertex cover in this case will be $mC_1 + 2nC_2$. Let $K = \frac{C_2}{C_1}$. Therefore,

$$\frac{W_a}{W_{opt}} = \frac{mC_1}{2nC_2} + 1 = \frac{m}{2nK} + 1. \tag{5}$$

The question that arises here is: What is the maximum value that $\frac{m}{nK}$ can attain? From the inequality $Q(v, m, n) < Q(u, m, n)$ we obtain

$$K > \frac{(m+2n)(m+2)}{n(m+4n)} \tag{6}$$

For given values of m and n, the maximum value of $\frac{m}{nK}$ is asymptotically attained when

$$K = \frac{(m+2n)(m+2)}{n(m+4n)}.$$

Therefore, we may write

$$\max\left(\frac{m}{nK}\right) = \frac{m(m+4n)}{(m+2n)(m+2)} = \left(\frac{m}{m+2}\right)\left(1+\frac{2n}{2n+m}\right)$$

Clearly, for large values of m, $\frac{m}{m+2} \rightarrow 1$ and for $2n >> m$, $\frac{2n}{2n+m} \rightarrow 1$. Thus, $\max\left(\frac{m}{nK}\right)$ asymptotically reaches 2. From (5) we see that $\frac{W_a}{W_{opt}}$ asymptotically reaches 2.

**Case 3:** Q(v, m, n) = Q(u, m, n).

Either a u vertex or a v vertex may be chosen in the first iteration. If a u vertex is chosen, then it follows from (4) that a u vertex will also be chosen in the next m iterations. This subcase is then identical to case 2. Instead, if a v vertex is chosen, the corresponding w vertex will have degree 1. This will cause the other v vertex of the triangle to be chosen (by $H_1$). By (3), all v vertices will be chosen in the subsequent iterations. Thus, this subcase is identical to case 1.

Let us now compute the maximum value of $\frac{W_a}{W_{opt}}$ assuming all vertices had the same cost (K = 1). Again, the maximum value would be attained if all the u vertices were picked before the v vertices (as in case 2 above). Here $\frac{W_a}{W_{opt}} = \frac{m}{2n}+1$. Substituting K = 1 in (6) and simplifying, we get

$$4n^2 - n(4+m) - (m^2 + 2m) > 0.$$

Solving for n, we get

$$n > \frac{4+m+\sqrt{(4+m)^2+16(m^2+2m)}}{8}$$

$$= \frac{4 + m + \sqrt{17m^2 + 40m + 16}}{8}$$

The maximum value of $\frac{m}{n}$ is attained when n has its smallest value. Using similar arguments as before, we see that

$$\max\left(\frac{m}{2n}\right) = \frac{4m}{4 + m + \sqrt{17m^2 + 40m + 16}}$$

The right hand side of the above equality is a monotonically increasing function in m.

$$\lim_{m\rightarrow\infty} \frac{4m}{4 + m + \sqrt{17m^2 + 40m + 16}} = \frac{4}{(1 + \sqrt{17})} = 0.7808.$$

Therefore the maximum value of $\frac{W_a}{W_{opt}}$ when K = 1 is approximately 1.7808.

The above discussion does not in any way imply that 2 is an upper bound on the performance of our algorithm. However, we have not been able to discover a higher ratio so far.

### 5.3. The Expected Size of the Smallest Vertex Cover

**Definition:** A graph G on n vertices is said to belong to the $G_{n, p}$ class of random graphs if each of the $\frac{n(n-1)}{2}$ edges is included in G independently and with probability p, $0 < p < 1$. □

It has been shown in Chapter 11 of [Bollobas85] that the expected size of the largest clique (denoted by $r_0$) for the $G_{n, p}$ class is very accurately predictable. In fact it can be bracketed between two consecutive integers for large random graphs. To calculate the expected size of the largest independent set, we replace p by (1-p) in the expression for $r_0$. Let $\alpha(G)$ denote the size of the largest independent set for the $G_{n, p}$ class of graphs. Assuming unit cost vertices, the size of the smallest vertex cover is then given by $n - \alpha(G)$.

We will outline the derivation of the expression for $r_0$ here. We will use the same techniques for deriving the expected size of the smallest vertex cover in random hypergraphs with unit cost vertices in Section 5.6.

Consider subsets of r vertices in a random graph on n vertices. There are C(n, r) such subsets[21]. Associate the 0-1 random variable $X_i$ with each r-subset. $X_i$ is defined as follows:

$$X_i = \begin{cases} 1 & \text{if the ith r–subset forms a clique.} \\ 0 & \text{if the ith r–subset does not form a clique.} \end{cases}$$

For an r-subset to form a clique, all the corresponding C(r, 2) edges must be present. Therefore

$$X_i = \begin{cases} 1 & \text{with probability } p^{C(r, 2)} \\ 0 & \text{otherwise} \end{cases}$$

Therefore, the expected value of $X_i$, denoted by $E(X_i)$, is equal to $p^{C(r, 2)}$. Let $Y = \sum_{i=1}^{C(n, r)} X_i$. Clearly, Y denotes the number of cliques of size r. The expected value of Y, denoted by E(Y, r), is equal to

[21] C(n, r) denotes the number of ways of choosing r items from n items and is given by $\frac{n!}{r!(n-r)!}$.

$$\sum_{i=1}^{C(n, r)} E(X_i) = C(n, r) * p^{C(r, 2)} \quad (1)$$

This is because, for any random variables, the mean of the sum is equal to the sum of the means. The value of E(Y, r) increases initially for small values of r and then falls monotonically with increasing values of r. The value of $r_0$ is equal to that value of r such that E(Y, r) = 1. This is because, for values of $r > r_0$, the expected number of cliques of size r is less than 1 and hence it is very unlikely that a clique of size of $r > r_0$ will occur. Let $r_1$ be such that $E(Y, r_1) > 1 > E(Y, r_1 + 1)$. Clearly, $r_1 < r_0 < r_1 + 1$. For large values of n, we can replace n! with its Stirling approximation[22] in (1) and solve for $r_0$. It is shown in [Bollobas85] that, for large random graphs, $r_0$ is given by the following equation:

$$r_0 = 2\log_b n - 2\log_b \log_b n + 2\log_b(\frac{e}{2}) + 1 + o(1)$$

where $b = \frac{1}{p}$.

Since we do not have any analytical bound on the performance of our algorithm, we used the above formula as a standard of comparison.

### 5.4. The Experiments on Random Graphs

Before we describe our experiments, we will present the algorithm we used to generate random graphs. For all practical purposes, generating truly random graphs in the $G_{n, p}$ class is impossible as we do not have an access to a perfect random number generator. Instead, we used the random generator function called **random** provided by the 4.3 UNIX Operating System. Each invocation of the random function returns an integer in the range 0 through M where $M = 2^{31} - 1$. We now present our algorithm for the generation of random graphs in pseudo code. Let N denote the total number of edges in a complete graph on n vertices. $N = \frac{n(n-1)}{2}$. Assume that these edges are enumerable in some order.

---

[22]For large values of n, the Stirling approximation for n! is given by: $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$

**Algorithm**

```
for i = 1 to N do
rand = random( )
If (rand < M * p)
     then include the ith edge in the graph
     else discard the ith edge.
end_for
```

**End Algorithm**

We conducted four sets of experiments. We describe each of them below and present the results graphically.

**The First Set:** In this set of experiments we generated random graphs with different values of n and p. The values of n were picked from the set {25, 50, 75, 100}. The value of p was varied from a minimum of 0.01 to a maximum of 0.5. For a specific pair of values of n and p, we generated 100 random graphs. All vertices had unit costs. We ran the Bar-Yehuda algorithm on each of the 100 graphs. We then ran our approximation algorithm on each of the 100 graphs. The average value of the vertex cover produced (by each of the two algorithms) over these 100 graphs was calculated. In Graph 5.1, we compare the averages obtained by the two algorithms with the expected size of the minimum vertex cover. Graph 5.1 has 4 sets of 3 curves each. Each set of curves corresponds to a particular value of n and are indicated by a different line type and also by the value of n below them. The lowest curve in each set gives the expected value of the vertex cover. The curve in the middle gives the average value of the vertex cover obtained by our approximation algorithm. The highest curve in each set gives the average value of the vertex cover obtained by the Bar-Yehuda algorithm.

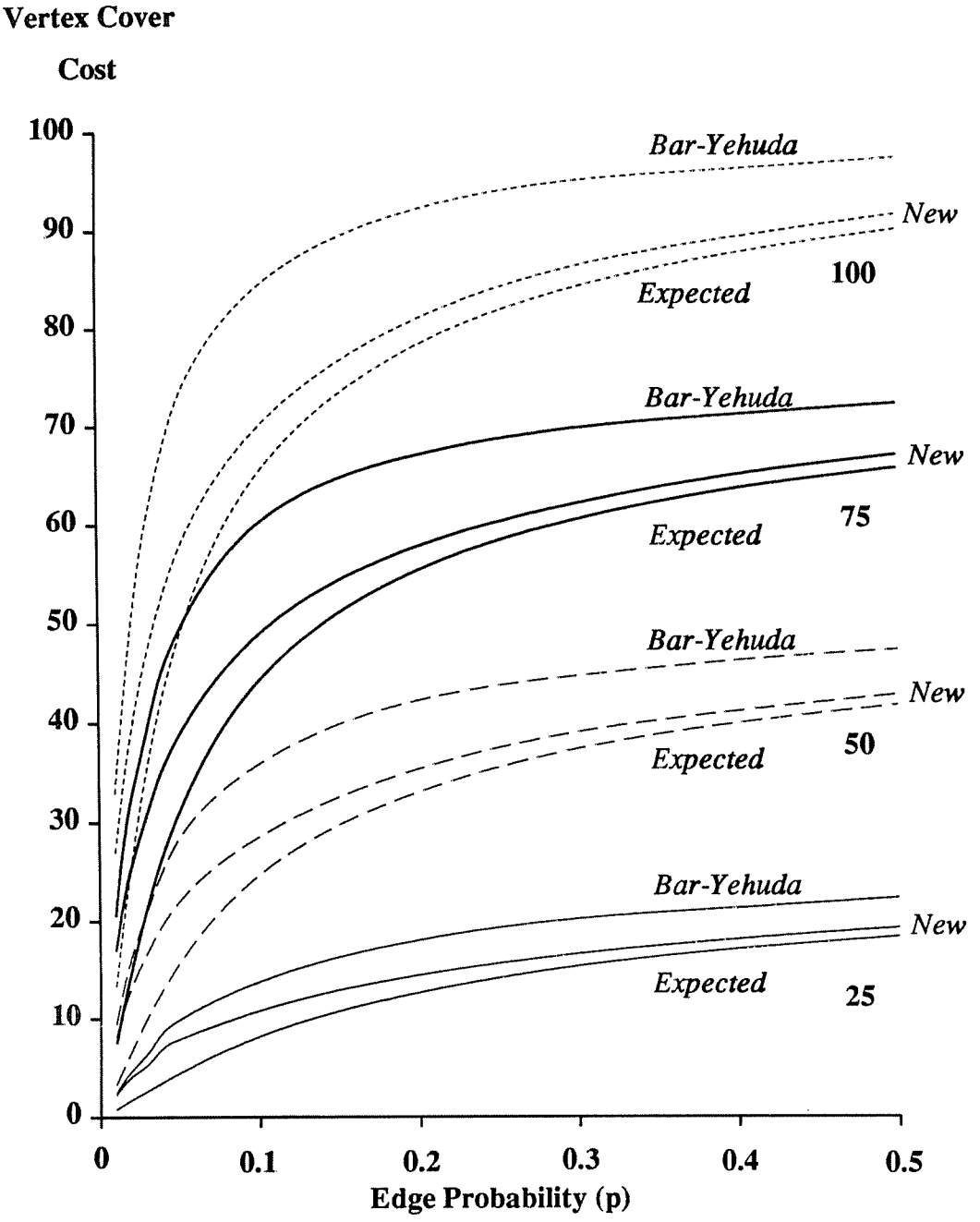
**The Second Set:** In the set of experiments discussed above the vertices had unit costs. In this set the weights were 100 numbers randomly distributed between 1 and 100. Thus, every vertex had a preassigned weight depending on its ordinal position. In Graph 5.2, we compare the averages obtained by the two algorithms. Graph 5.2 has 4 sets of 2 curves each. The lower curve in each set gives the average value of the vertex cover obtained by our approximation algorithm. The higher curve in each set gives the average value of the vertex obtained by the Bar-Yehuda algorithm.

**The Third Set:** Here we generated random graphs on a small number of vertices so that we could run the optimal algorithm on the graphs. We picked n from the set {10, 14, 18}. The value of p was varied from a minimum of 0.1 to a maximum of 0.5. For each pair of values of n and p, we generated 100

random graphs. In addition to running the two approximation algorithms on these graphs, we also obtained the optimal solution by running the exponential algorithm on each of the 100 graphs. The vertices had unit costs. In Graph 5.3, we compare the averages obtained by the two approximation algorithms with that of the average obtained from the optimal algorithm. Graph 5.3 has 3 sets of 3 curves each. The lowest curve in each set gives the average value of the vertex cover obtained by the optimal algorithm. The middle curve in each set gives the average value of the vertex cover obtained by our approximation algorithm. The highest curve in each set gives the average value of the vertex obtained by the Bar-Yehuda algorithm. Notice that in all the 3 sets, the optimal and the new algorithms were almost identical in performance.

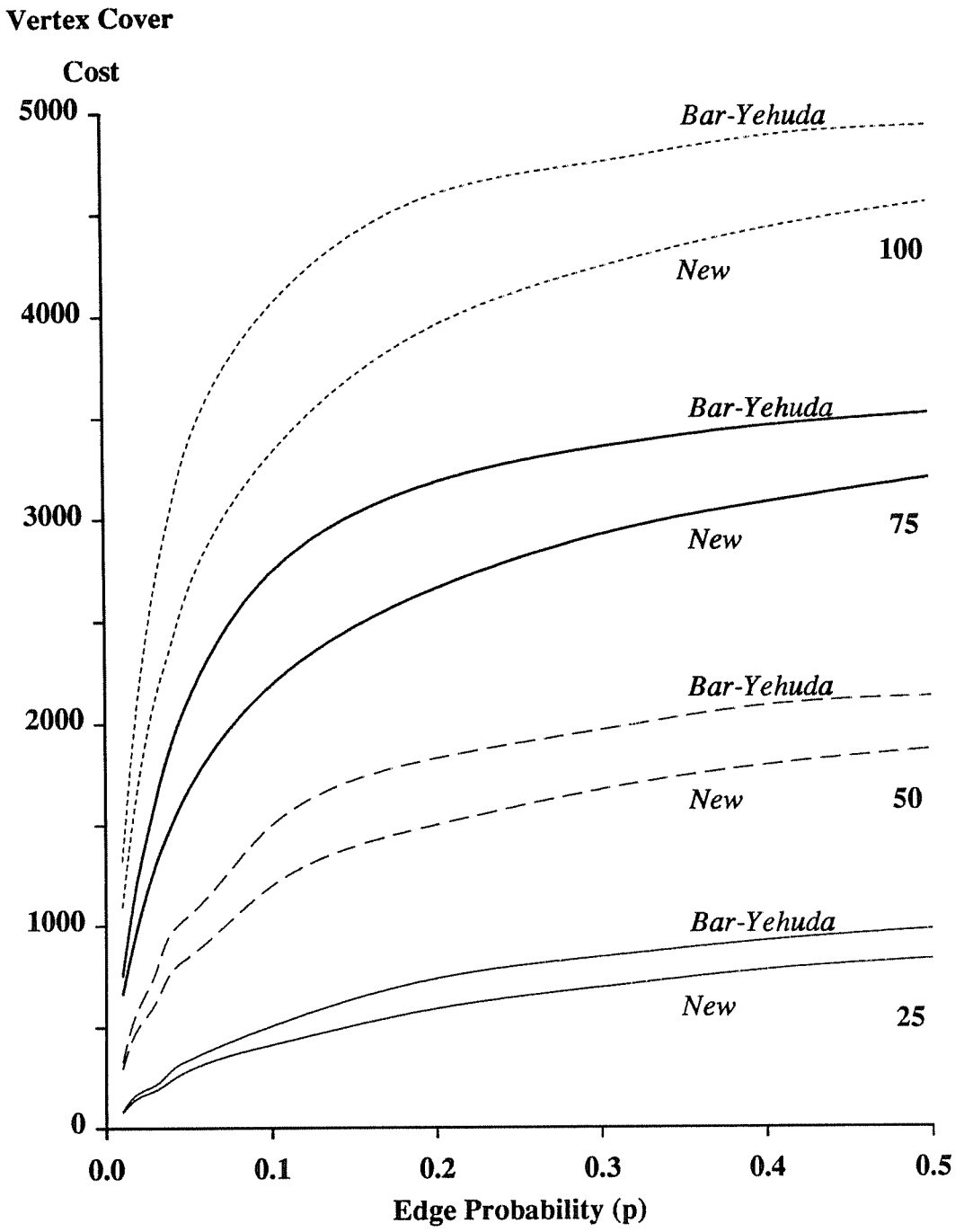
**The Fourth Set:** This set was identical to the third set except that the weights of the vertices were randomly distributed between 1 and 100. As in Set 2, every vertex had a preassigned weight depending on its ordinal position. The results are presented in Graph 5.4. Graph 5.4 has the same format as Graph 5.3. As in Graph 5.4, the Optimal and New curves in each set are very close to each other.

From the experiments, it is evident that, on the average, our approximation algorithm performs better than the Bar-Yehuda algorithm for random graphs over a wide range of densities. From the third and fourth set of experiments, we see that for smaller graphs ($\leq 20$ vertices), our approximation algorithm gives vertex covers that are very close to the optimal vertex covers. Unfortunately it becomes infeasible to compare how well our approximation algorithm performs for larger graphs as running the optimal algorithm on larger graphs becomes prohibitively expensive.



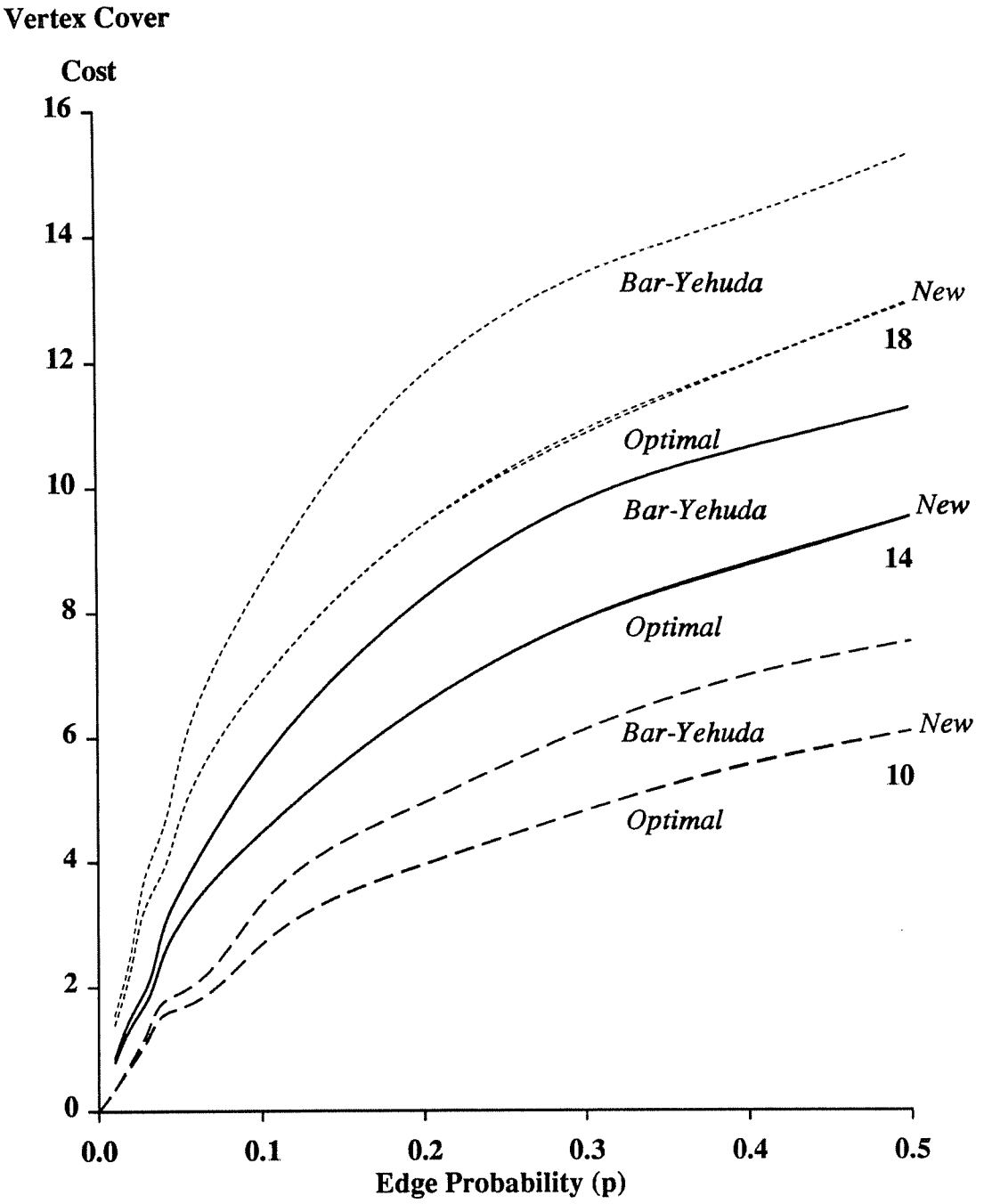

**Graph 5.1.**

**(Unit Cost Vertices)**



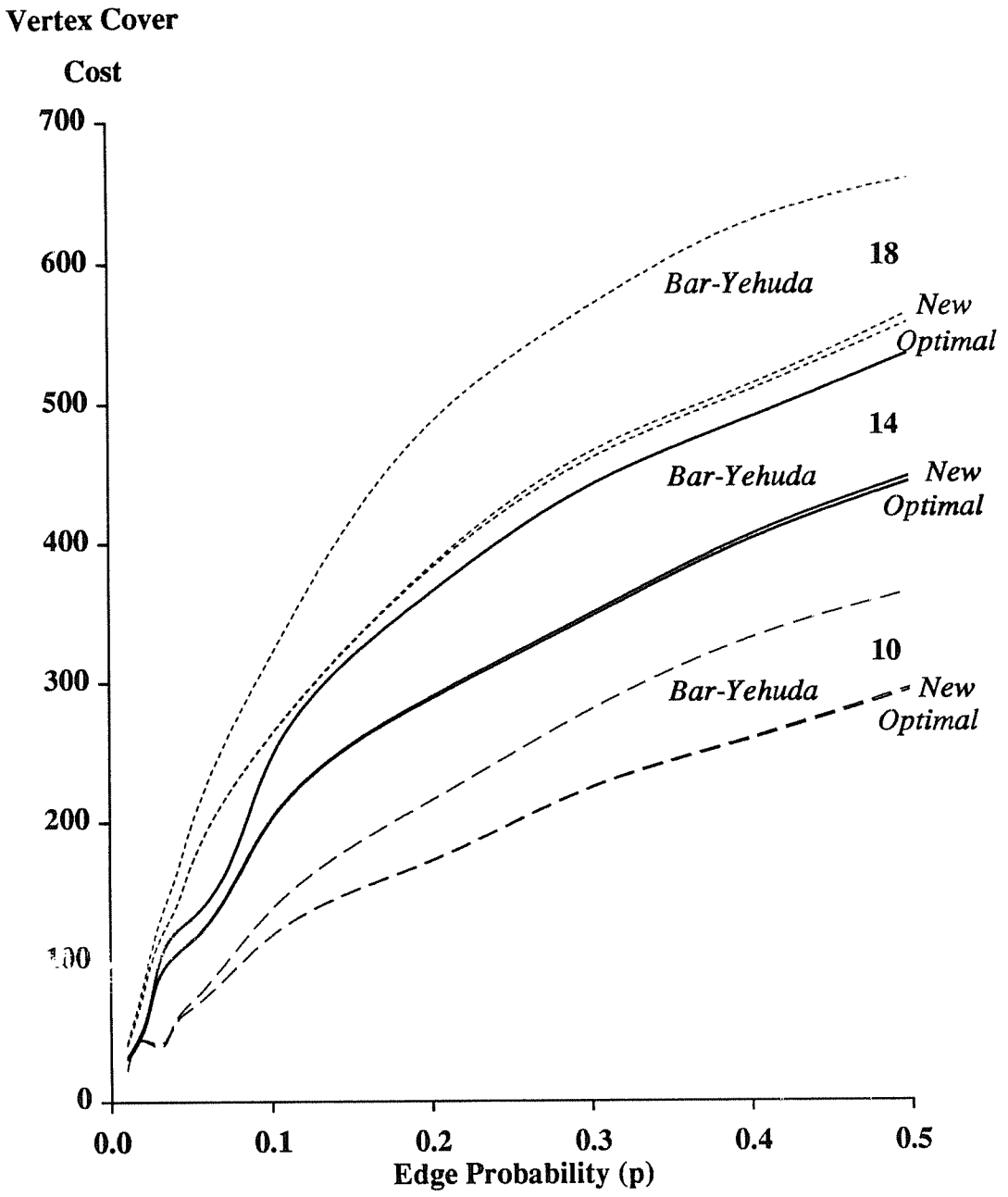

**Graph 5.2.**

**(Random Cost Vertices)**




**Graph 5.3.**

**(Unit Cost Vertices)**




**Graph 5.4.**

**(Random Cost Vertices)**

## 5.5. The Approximation Algorithm for Hypergraphs

We saw in Section 4.8 that an optimal vertex cover can be found efficiently for 2-dimensional (2-partite) hypergraphs. Therefore, we will focus our attention on higher dimensional hypergraphs[23]. The approximation algorithm for simple graphs consisted of two parts, viz., $H_1$ and $H_2$. In what follows, we discuss the extension of these parts to hypergraphs. We will denote the extensions of $H_1$ and $H_2$ to hypergraphs by $HY_1$ and $HY_2$ respectively. The extension of the heuristic, $H_2$ to $HY_2$, is straight-forward and we will present it first.

Let F(j) be the set of vertices incident on the jth hyperedge, and S(i) be the set of hyperedges incident on the ith vertex. The multi-set[24] (duplicates possible) of neighbors of a vertex v, denoted by N(v), is given by:

$$N(v) = \bigcup_{e \in S(v)} (F(e)) - v$$

The above formula agrees with our intuitive sense of what the neighbors of a vertex in a hypergraph are. We calculate Q(v) for every vertex in the hypergraph, where Q(v) is given as before by:

$$Q(v) = \sum_{v_i \in N(v)} \frac{cost(v_i)}{degree(v_i)} - cost(v)$$

The vertex with the maximum Q value is included in the cover. Ties may be broken arbitrarily. Once a vertex is chosen in the cover, we remove that vertex and all the hyperedges incident on it from the hypergraph.

We now present the extension of $H_1$ to $HY_1$. As in the case of simple graphs, the idea of $HY_1$ is to avoid including unit degree vertices in the vertex cover whenever possible. Let us illustrate this with an example. Consider the 3-dimensional hypergraph in Figure 5.4. The vertices are numbered as shown and the four hyperedges are 1--4--8, 2--4--7, 3--4--6, and 2--5--7. Let cheapest_vertex[j] denote the vertex with the cheapest cost on the jth hyperedge. If

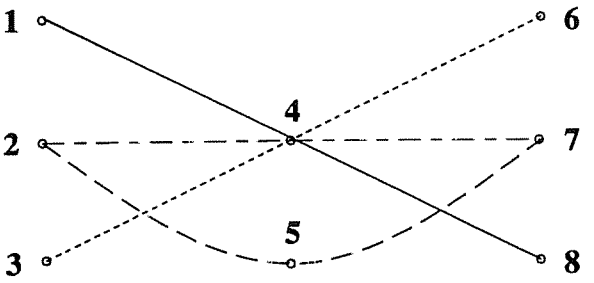
$$cost(4) \leq cost(cheapest\_vertex[1\text{--}4\text{--}8]) + cost(cheapest\_vertex[3\text{--}4\text{--}6]),$$

we know that vertex 4 will belong to the optimal vertex cover. It would not be profitable to include any of

---

[23]Our algorithm may, however, be applied to a 2-dimensional hypergraph as well.

[24]The neighbors of vertex 2 in the hypergraph shown in Figure 5.4 are {4, 7, 5, 7}. Vertex 7 can be reached from vertex 2 via two hyperedges.




$$\text{cost}(1) = \text{cost}(3) = \text{cost}(4) = 1.$$
$$\text{cost}(5) = \text{cost}(6) = \text{cost}(8) = 1.$$
$$\text{cost}(2) = 2, \text{cost}(7) = 10.$$

$$Q(1) = Q(3) = Q(6) = Q(8) = \frac{1}{3}.$$
$$Q(2) = 9\frac{1}{3}, Q(4) = 9.$$
$$Q(5) = 5, Q(7) = 1\frac{1}{3}.$$

**Figure 5.4**

the unit degree vertices (1, 3, 6, 8) on these two edges in the cover. Consider the general situation where n edges all pass through a common vertex v such that every vertex on any of these n edges, with the exception of v, (viz., the neighbors of v) is of degree 1. Then, if

$$\text{cost}(v) \leq \sum_{i=1}^{n} \text{cost}(\text{cheapest\_vertex}[i]),$$

we must include v in the vertex cover. Once v is included in the vertex cover, we remove vertex v from the hypergraph along with all the edges incident on it. On the other hand, if

$$\text{cost}(v) > \sum_{i=1}^{n} \text{cost}(\text{cheapest\_vertex}[i]),$$

and these are the only edges in the hypergraph, then $\bigcup_{i=1}^{n}(\text{cheapest\_vertex}[i])$ is clearly the optimal vertex cover.

The algorithm for hypergraphs is identical to the one presented for graphs in Section 5.1 and is reproduced here.

**Algorithm**

```
while (edges exist in the hypergraph)
begin
    while (H1 can be applied)
    begin
        Apply H1.
        Remove the corresponding vertex and the edges incident on it.
    end while

    if (edges exist in the hypergraph)
    begin
        Calculate Q(v) for every vertex in the hypergraph.
        Include v with maximum Q(v) in the cover.
        Remove v and the edges incident on it.
    end if
end while
```

**End Algorithm**

We will discuss the time complexity of the algorithm for hypergraphs at the end of the section.

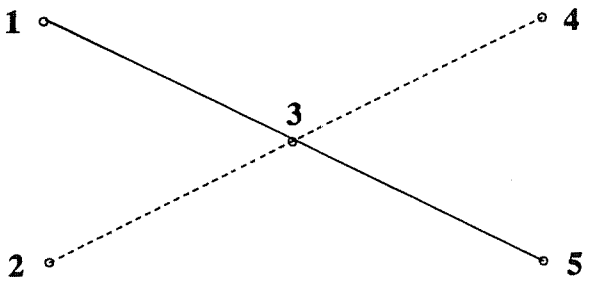
As was the case with simple graphs, we must point that it is important to apply $HY_2$ only if $HY_1$ cannot be applied. Let us illustrate this with two examples.

**Example 1**: First, reconsider the hypergraph in Figure 5.4. Let cost(1) = 1, cost(2) = 2, cost(3) = cost(4) = cost(5) = cost(6) = 1, cost(7) = 10, and cost(8) = 1. Applying $HY_1$, we see that it would be optimal to cover edges 1--4--8 and 3--4--6 with vertex 4. After removing vertex 4 and the edges incident on it, the only edge left is 2--5--7. $HY_1$ can be applied again to the edge[25] 2--5--7 which will be covered with vertex 5. Thus, the cost of the vertex cover obtained is cost(4) + cost(5) = 2.

On the other hand, if $HY_2$ is applied first, we will get the following Q values: Q(1) = Q(3) = Q(6) = Q(8) = $\frac{1}{3}$, Q(2) = $9\frac{1}{3}$, Q(4) = 9, Q(5) = 5, and Q(7) = $1\frac{1}{3}$. This would cause vertex 2, with cost 2, to be picked. Applying $HY_1$ or $HY_2$ in the next iteration will cause vertex 4 to be picked. The total cost of the vertex cover is cost(4) + cost(5) = 1 + 2 = 3. Thus, in general, applying $HY_2$ before $HY_1$ can lead to vertex covers of higher cost. □

**Example 2**: Consider the 3-dimensional hypergraph in Figure 5.5. The cheapest vertex on edge 1--3--5 is 1 and the cheapest vertex on edge 2--3--4 is 4. Since these are the only two edges in the hypergraph, and since cost(3) > cost(1) + cost(4), the optimal vertex cover is the set {1, 4}. By applying

[25]Notice that $HY_1$ and $HY_2$ will always choose the cheapest vertex on a single isolated edge.




**cost(1) = 1, cost(2) = 2, cost(3) = 3, cost(4) = 1, cost(5) = 2.**

**Q(1) = Q(4) = 2.5.**

**Q(2) = Q(5) = 0.5.**

**Q(3) = 3.**

**Figure 5.5**

$HY_1$, we will get this optimal solution. However, if we apply $HY_2$ first, we will obtain the Q values shown in Figure 5.5. Vertex 3 has the maximum Q value, resulting in a higher cost vertex cover. □

Thus, the lemma presented in Section 5.1 for simple graphs cannot be extended to hypergraphs.

We will conclude this section with a couple of observations that may lead to lower cost vertex covers than those yielded by our approximation algorithm.

**Observation:** Let k be the number of edges in the hypergraph. The set S given by $\bigcup_{i=1}^{k}$(cheapest_vertex[i]) is a cover. If the cost of the vertex cover obtained by our algorithm is higher than the cost of S, S is obviously a better cover. We will call this observation $HY_3$. □

The next observation applies only to the restricted class of hypergraphs (as defined in Section 4.5) which we are primarily interested in.

**Observation:** The set of vertices in any of the disjoint partitions forms a vertex cover. Let S denote the cheapest vertex partition. If the cost of the vertex cover obtained by our algorithm is higher than the cost of S, S is obviously a better cover. We will call this observation $HY_4$. □

We now discuss the worst case time complexity of our algorithm when applied to hypergraphs in. We assume that the m-dimensional hypergraph is stored in the form of the S and F sets. F(j) is the set of

vertices incident on the jth edge, and S(i) is the set of edges incident on the ith vertex (degree(i) = |S(i)|). Let n be the total number of vertices in the hypergraph and $S_{max} = \max(|S_i|)$, $1 \le i \le n$.

To determine if $H_1$ is applicable, we need to look at the neighbors of a vertex v given by

$$N(v) = \bigcup_{e \in S(v)} (F(e)) - v$$

Since |F(j)| = m for every edge in the hypergraph, the complexity of finding N(v) is O(|S(v)| * m). Therefore, the complexity of finding the neighbors of all the n vertices is $O(n * m * S_{max})$.

Similarly, to apply $H_2$, we need to find the neighbors of all vertices. In each pass through the body of the outer while loop at least one vertex is removed. In a complete hypergraph[26] one can remove a maximum of (n - m + 1) vertices before all edges are covered. Thus, the worst case time complexity of the algorithm is $O(n^2 * m * S_{max})$.

### 5.6. The Expected Size of the Smallest Vertex Cover

**Definition:** An m-dimensional hypergraph H with $n_i > 0$ vertices in the ith partition, $1 \le i \le m$, belongs to the $H_{n_1, n_2, ..., n_m, p}$ class of random hypergraphs if each of the $\prod_{i=1}^{m} n_i$ edges is chosen independently and with probability p, $0 < p < 1$. □

As we saw in Section 5.3, it was possible to compute the expected size of the minimum vertex cover for random graphs with unit cost vertices. Using the same techniques as those of in Section 5.3, we will derive the expected size of the smallest vertex cover for the $H_{n_1, n_2, ..., n_m, p}$ class of hypergraphs.

**Definition:** An **independent set** in an m-dimensional hypergraph consists of $m_i$ vertices from each partition ($0 \le m_i \le n_i$) such that there are no hyperedges among these vertices. □

The complement of the vertices in the independent set forms a vertex cover. Let $\alpha(H) = \sum_{i=1}^{m} m_i$ denote the size of the largest independent set. The size of the smallest vertex cover is then equal to $\sum_{i=1}^{m} n_i - \alpha(H)$.

---

[26] A complete m-dimensional hypergraph with $n_i$ vertices in each partition is one in which all the possible $\prod_{i=1}^{m} n_i$ edges are present.

The number of ways we can choose $m_i$ vertices from $n_i$ vertices is $C(n_i, m_i)$. The total number of ways of choosing $m_i$ vertices from $n_i$ vertices among all the m partitions is therefore equal to $\prod_{i=1}^{m} C(n_i, m_i)$. The maximum number of hyperedges among a specific subset of vertices, consisting of $m_i$ vertices from each partition, is $\prod_{i=1}^{m} m_i$. Let $K = \prod_{i=1}^{m} m_i$. For this subset of vertices to be independent, none of the K hyperedges must occur. The probability that the subset is an independent set is $q^K$ where $q = 1 - p$.

Let Y denote the number of independent sets of size r. Using the techniques of Section 5.3, we can see that E(Y, r), the expected number of independent sets of size r is given by

$$E(Y, r) = \left(\sum_{i=1}^{m} (m_i) = r\right) \prod_{i=1}^{m} C(n_i, m_i) q^{m_i} \qquad (1)$$

where $0 \le m_i \le n_i$, $1 \le i \le m$. The value of E(Y, r) increases initially for small values of r and then falls monotonically with increasing values of r. The value of $\alpha(H)$ is equal to that value of r such that E(Y, r) = 1. This is because, for values of $r > \alpha(H)$, the expected number of cliques of size r is less than 1. Let $r_1$ be such that $E(Y, r_1) > 1 > E(Y, r_1 + 1)$. Clearly, $r_1 < \alpha(H) < r_1 + 1$. The size of the smallest vertex cover is then given by $\min \{n_1, n_2, ..., n_m, \sum_{i=1}^{m} n_i - \alpha(H)\}$. Since (1) is fairly complex, computing the precise value of $\alpha(H)$ seems infeasible. We therefore approximate $\alpha(H)$ to be equal to $r_1 + 0.5$. This guarantees that the worst case error in estimating the value of $\alpha(H)$ will never exceed 0.5. We have used this formula as the standard of comparison in some of the experiments.

### 5.7. The Experiments on Random Hypergraphs

The Algorithm for generating random hypergraphs is identical to the one for generating random graphs (presented in Section 5.4) except that in the case of hypergraphs, N would be equal to $\prod_{i=1}^{m} n_i$.

As in the case of random graphs, we performed four sets of experiments on random hypergraphs. In addition to choosing the number of vertices ($n = \sum_{i=1}^{m} n_i$) and the edge probability p, we also had to vary m, the number of dimensions of the hypergraph. Also, given n and m, we still had to decide on how to parti-

tion n vertices among each of the m dimensions. Given a certain pair of values of n and m, we can compute the $n_i$'s randomly such that $n = \sum_{i=1}^{m} n_i$. Computing the $n_i$'s this way gave rise to some $n_i$'s that were much smaller than some others. In such a case it is very likely that a vertex in a smaller partition will be in the optimal vertex cover. When all vertices have the same cost, $HY_2$ will also cause such a vertex to picked since it will have a large number of neighbors. In order to test our approximation algorithm more fairly, we decided to make all the $n_i$'s equal to $\frac{n}{m}$. We now describe the four sets of experiments and present the results graphically.

**The First Set:** We varied m from 2 through 5. When m = 2, we let n = 22, 28, and 34. With m = 3, we used n = 21, 27, and 33. When m = 4, we picked n from the set {20, 24, 32}. With m = 5, we let n = 20, 25, and 30. Thus, $\frac{n}{m}$ was always an integer. For every pair of values of m and n, we varied p from a minimum of 0.01 to a maximum of 0.5. For every (n, m, p) triple, we generated 100 hypergraphs with unit cost vertices. We ran the Bar-Yehuda algorithm and our approximation algorithm on each of the 100 hypergraphs. The average value of the vertex cover produced by each of the two algorithms was calculated over these 100 hypergraphs. We present the results of the first set of experiments graphically in Figure 5.6. Figure 5.6 consists of four graphs. Each graph presents the results of the experiments on hypergraphs with a different number of partitions. In all the graphs in this section, we present the results for a specific value of n for each value of m. This was done in order to prevent overcrowding in the graphs. Each graph in Figure 5.6 has three curves. The lowest curve gives the expected value of the smallest vertex cover. The highest curve gives the average cost of the vertex cover obtained by the Bar-Yehuda Algorithm, while the middle curve gives the average cost of the vertex cover obtained by our approximation algorithm.

**The Second Set:** In the first set of experiments the vertices were of unit costs. In this set the weights were numbers randomly distributed between 1 and 100. Every vertex had a preassigned weight depending on its ordinal position. In Figure 5.7, we compare the averages obtained by the two algorithms graphically. Each graph has two curves. The lower curve gives the average obtained by our approximation algorithm, while the upper curve gives the average as obtained by the Bar-Yehuda Algorithm.

There are a couple of observations we can make from these two sets of experiments. First, our algorithm gives vertex covers that are significantly better than those obtained by the Bar-Yehuda Algorithm over a wide range of densities and for a varying number of partitions. Second, after the density increases beyond a certain point, the cost of the vertex cover obtained by our algorithm remains the same. The density at which the 'flattening' occurs is smaller for higher dimensional hypergraphs. The flattening occurs at higher densities because it becomes optimal to cover the edges with vertices of the cheapest partition, rather than with a few vertices from each partition. For the same density and the same total number of vertices, a hypergraph with a higher number of partitions will have more edges, provided the vertices are more or less equally divided among each partition. Because of this, the flattening occurs at much smaller densities for higher dimensional hypergraphs.

**The Third Set:** Here we generated random hypergraphs on a small number of vertices. We varied m from 2 through 5. When m = 2, we varied n from 10 to 18 in steps of 4. With m = 3, n was varied in steps of 3 from 9 through 15. When m = 4, n was varied from 8 through 16 in steps of 4. With m = 5, we picked n from the set {10, 15}. Thus, $\frac{n}{m}$ was always an integer. The value of p was varied from a minimum of 0.01 to a maximum of 0.5. For each (n, m, p) triple, we generated 100 random hypergraphs. In addition to running the two approximation algorithms on these hypergraphs, we also obtained the optimal solution by running the exponential algorithm on each of the 100 hypergraphs. The vertices had unit costs. In Figure 5.8, we compare the averages obtained by the two approximation algorithms with that of the average obtained from the optimal algorithm. The format of the graphs in Figure 5.8 is identical to the one for the graphs in Figure 5.6 except for the fact that the lowest curve in each graph gives the average vertex cover as obtained by the optimal algorithm. In all the four graphs in Figure 5.8, the optimal curve is indistinguishable from the curve corresponding to our approximation algorithm.

**The Fourth Set:** This set was identical to the third set except that the weights of the vertices were randomly distributed between 1 and 100. As in the second set of experiments, every vertex had a preassigned weight depending on its ordinal position. The results are presented in Figure 5.9.

In light of the fact that our heuristic gives near optimal vertex covers (Figures 5.8 and 5.9), it is unclear to us as to why there seems to be such a large disparity between the expected size of the smallest vertex cover and the vertex cover obtained by our heuristic (Figures 5.6). This disparity does not occur for

simple graphs (Graph 5.1).

We would also like to point out that the ratio of the running time of the new approximation algorithm to that of the running time of the Bar-Yehuda algorithm rarely exceeded 3 even when m = 5, n = 25, and p = 0.5. For smaller values of n, the corresponding ratio was very close to 1. Since the Bar-Yehuda algorithm is non deterministic, each run is likely to produce a vertex cover with a different cost. Running the Bar-Yehuda algorithm 3 times and taking the minimum cost over these three runs yielded a very marginal improvement in the performance of the Bar-Yehuda algorithm.

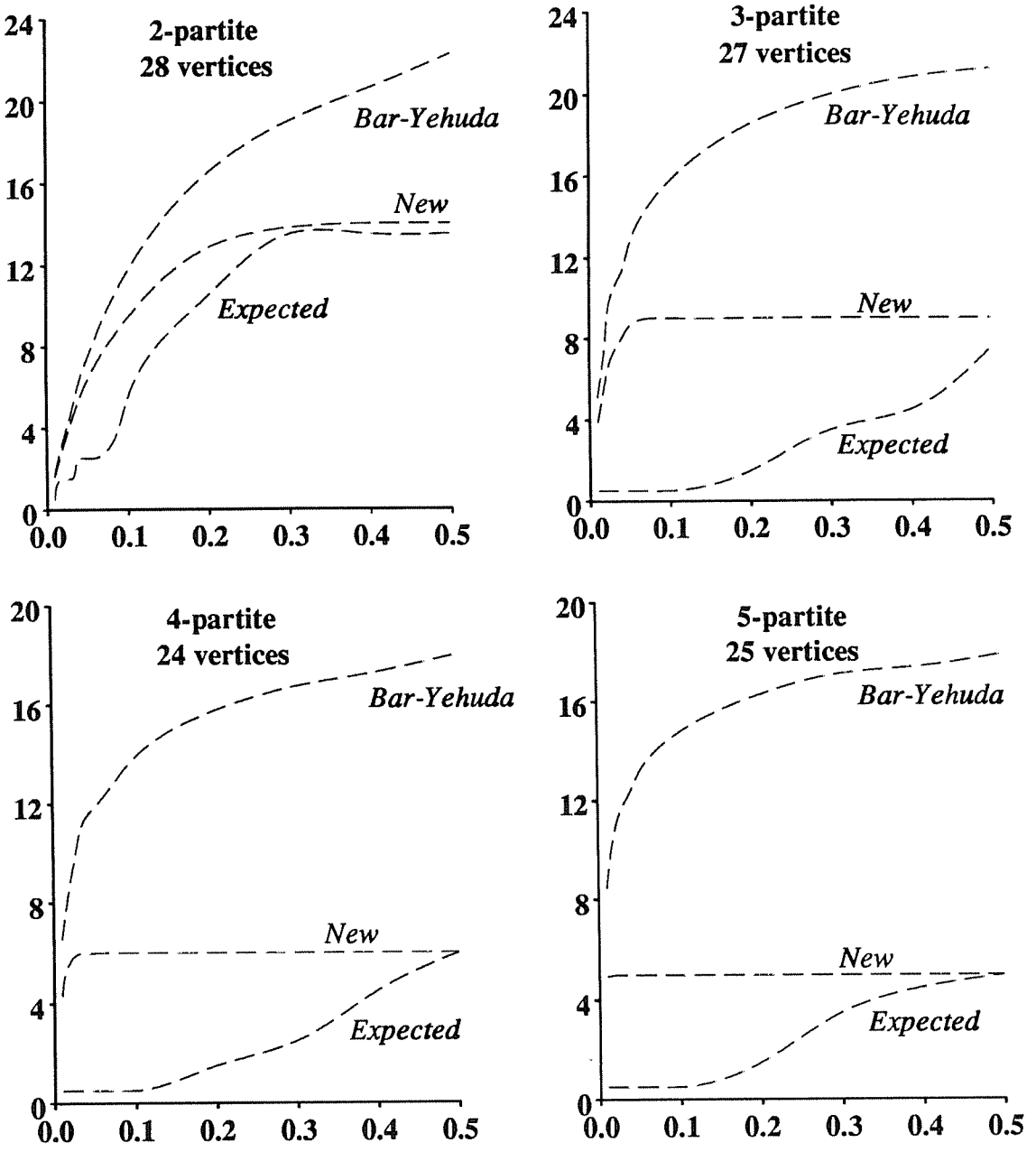



**Figure 5.6**

**(Unit Cost Vertices)**

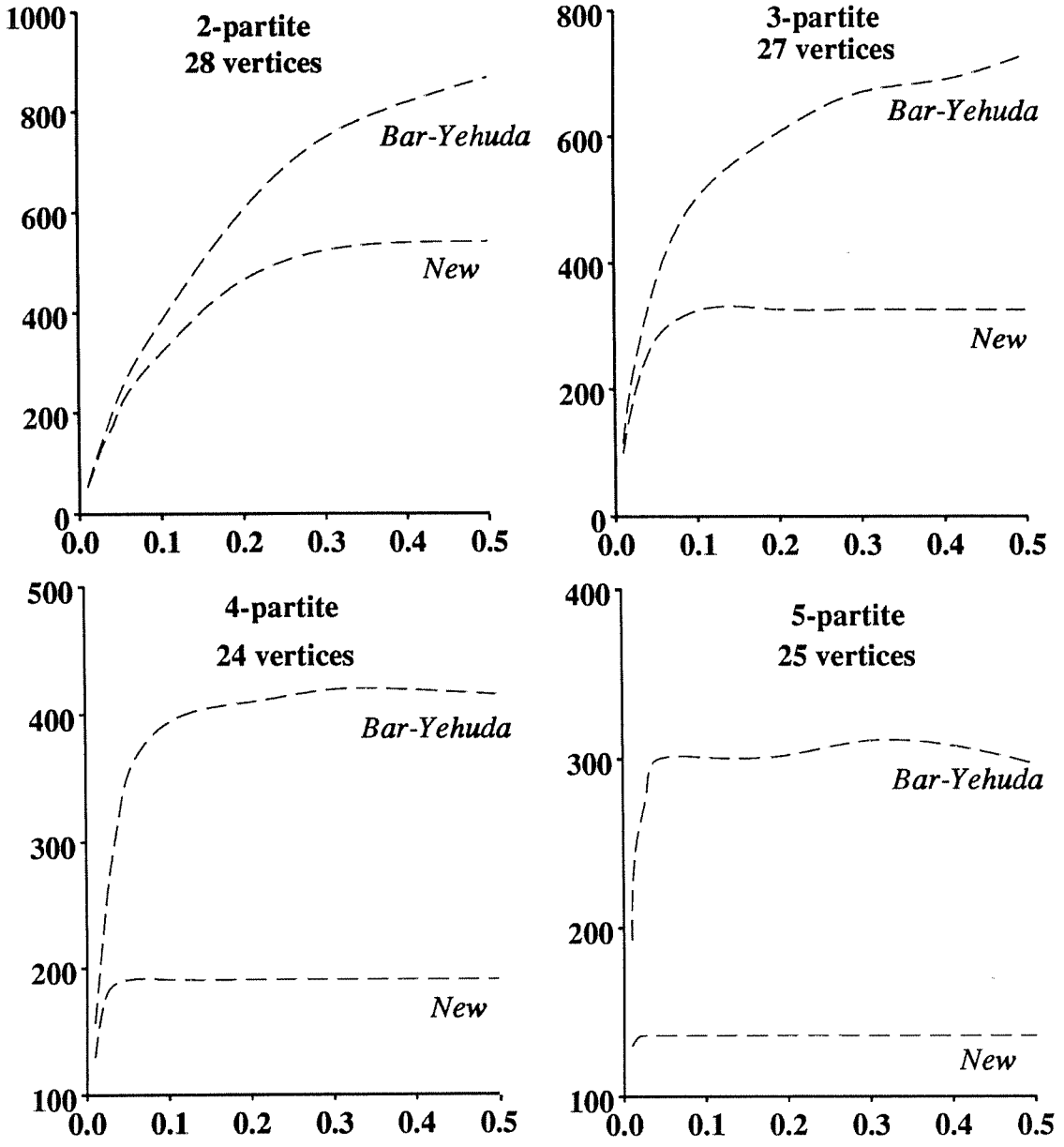



**Figure 5.7**

**(Random Cost Vertices)**

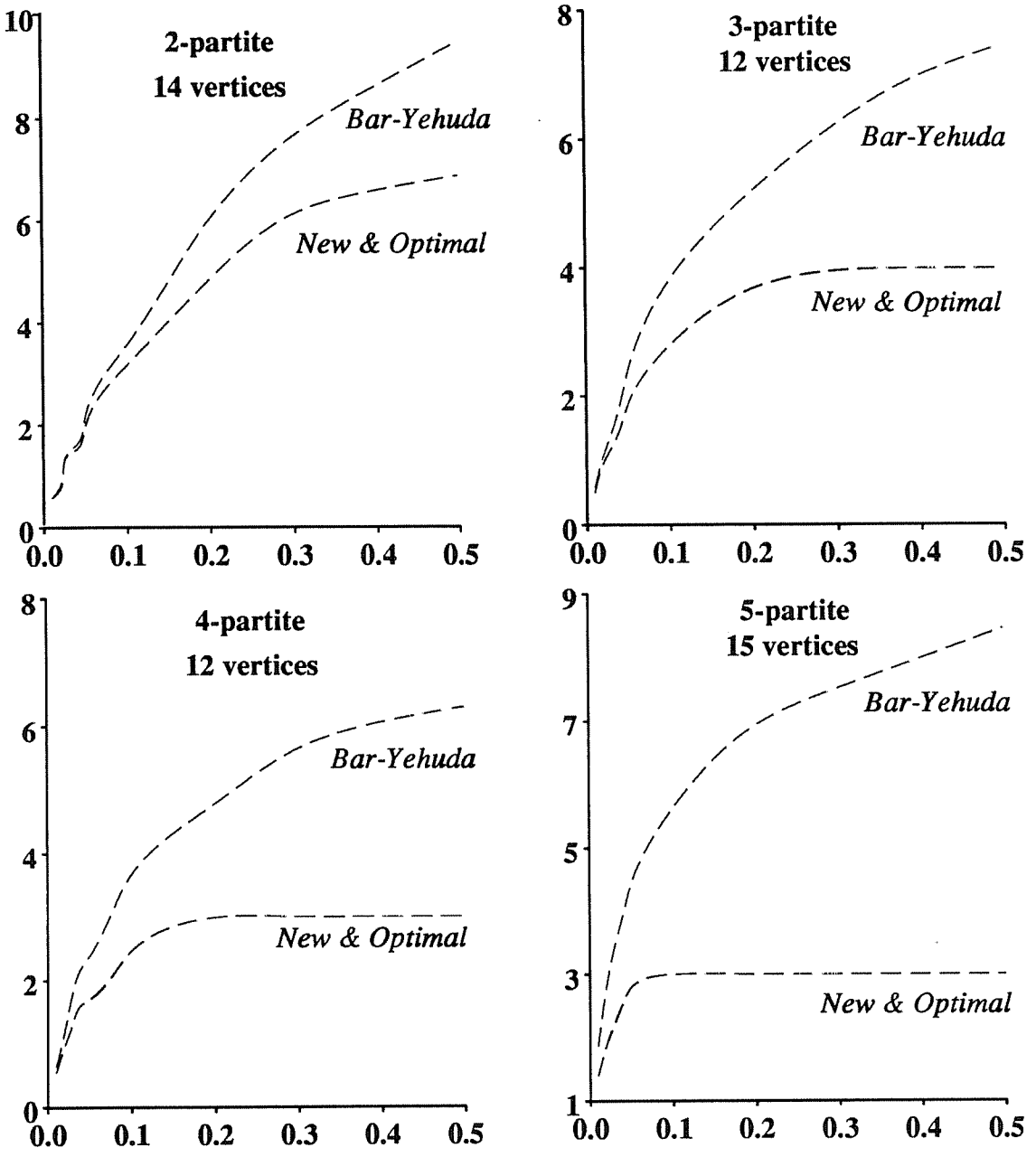



**Figure 5.8**

**(Unit Cost Vertices)**

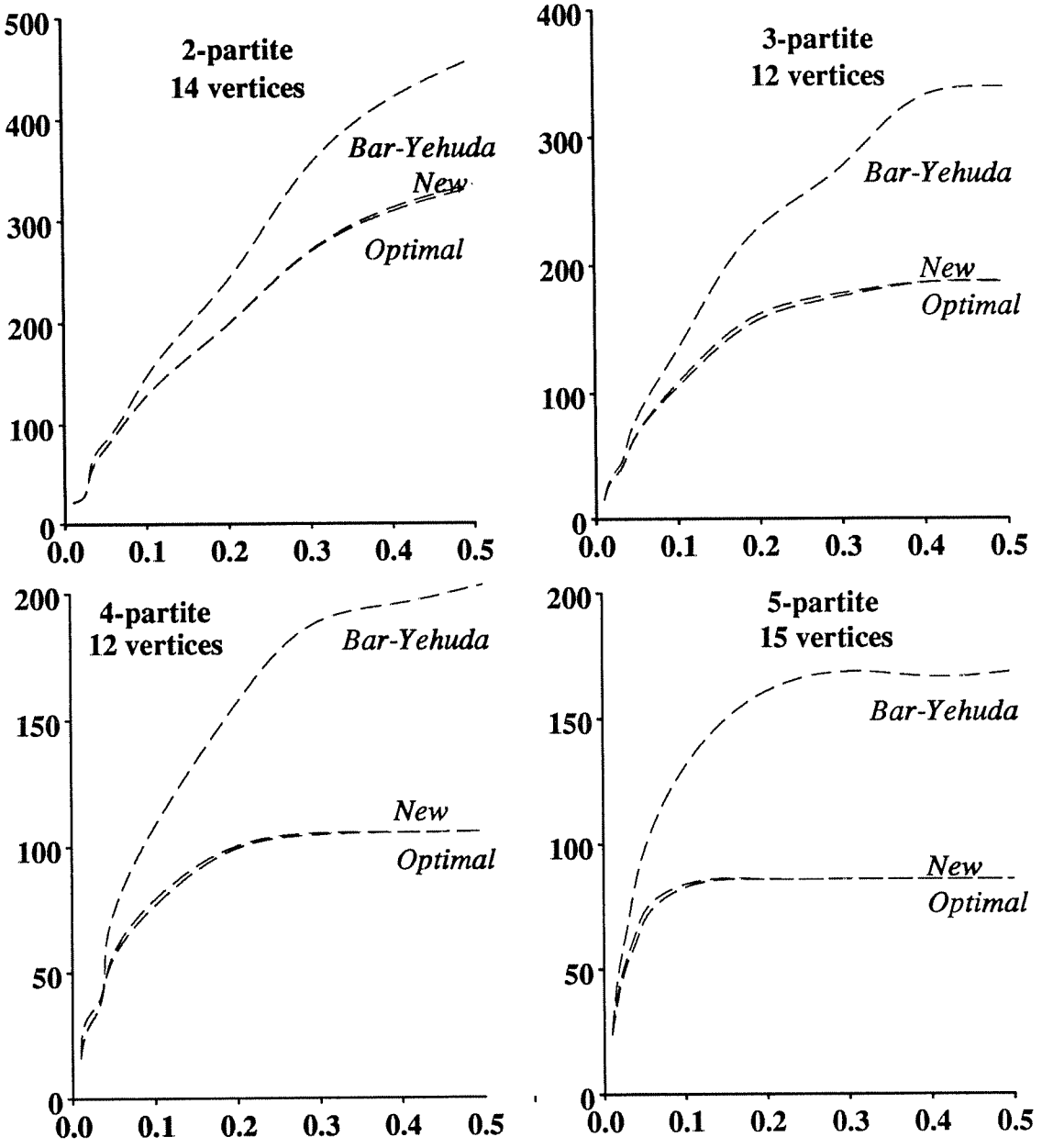



**Figure 5.9**

**(Random Cost Vertices)**

# CHAPTER 6

## MULTI-DIMENSIONAL HISTOGRAMS

Multi-dimensional queries commonly occur in databases dealing with geographical, image, and VLSI databases. A typical two dimensional query in a geographical database might involve finding all cities within certain latitudinal and longitudinal bounds. Several multi-dimensional index structures have been proposed in the literature for point data. KDB trees [Robinson81] and Grid files [Nievergelt84] are among the more popular ones. We described the techniques for using multi-dimensional indices in optimizing multi-dimensional queries in Section 4.11. However, there has been no work in designing multi-dimensional histograms to aid in the optimization process using these multi-dimensional index structures. In order for an optimizer to select an appropriate access path for a multi-dimensional query, fairly accurate selectivity estimates must be available to it. Selectivity estimates are also useful in determining appropriate join methods that follow the selections. The problem of building histograms on a single attribute has been thoroughly studied in [Shapiro84]. We will begin by describing the central concepts in [Shapiro84]. Traditionally, histograms were built such that each histogram bucket had the same width. It has since been shown that a way to control the maximum estimation error is to control the depth of each histogram and not its width. In other words, all histogram buckets must have the same depth and not the same width. As seen in [Shapiro84], it is necessary to sort the relation on the particular attribute in order to generate equi-depth histograms. The maximum selectivity estimation error can be arbitrarily reduced by increasing the number of equi-depth buckets.

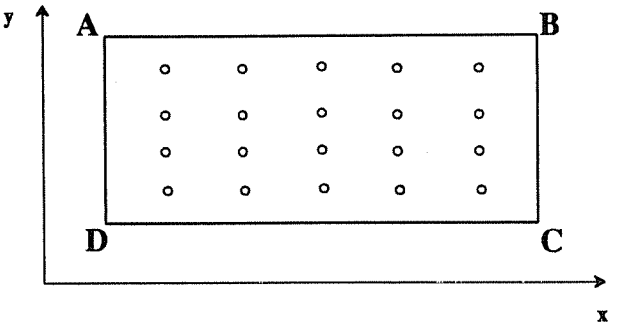
We plan to use the concept of equi-depth histograms in building multi-dimensional histograms for point data. In the next section we will describe the algorithm for generating multi-dimensional histograms of equal depth. A natural question that arises here is the following: Can we use d 1-dimensional histograms on each of the d attributes for estimating selectivity factors for multi-dimensional queries? It is easy to come up with examples where d 1-dimensional histograms will not be useful because 1-dimensional histograms cannot capture the notion of spatial locality of the tuples. The cost of building d 1-dimensional histograms is d times the cost of sorting the relation on a single attribute. One might expect that the cost of

building a d-dimensional histogram would be at least d times the cost of sorting the relation on a single attribute. As we will show, in our algorithm, the sorting cost of building a d-dimensional histogram is significantly less than the cost of sorting the relation d times. We will then present a main memory data structure for storing the histograms and discuss two schemes for estimating the number of tuples that will be retrieved by a given query. In subsequent sections, we will describe experiments and present results that show the efficacy of our histograms. We will also explore the usefulness of a sampling technique in generating histograms at a very low cost.

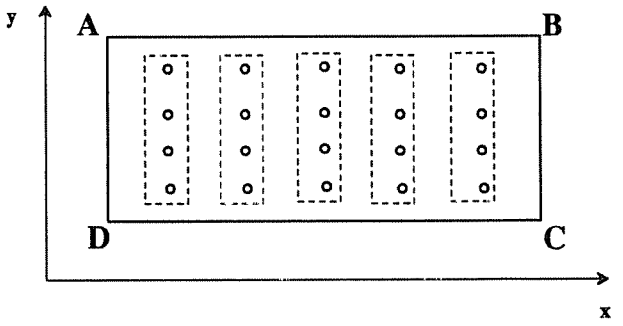
### 6.1. Generating Multi-Dimensional Histograms

Before we discuss our algorithm for generating multi-dimensional histograms, we must first describe what equi-depth multi-dimensional histograms will look like. Let us discuss this in the context of a 2-dimensional example. Assume a relation R with attributes x and y. Figure 6.1(A) shows a rectangle ABCD that represents the space of tuples of relation R. The points inside the rectangle represent the tuples. The problem of generating equi-depth histograms is equivalent to covering all the tuples in the tuple space with S rectangles such that each rectangle has the same number, viz., $\frac{|R|}{S}$ of tuples within it. Such rectangles are called equi-depth histograms or equi-depth buckets. We will hereafter use the terms bucket and histogram interchangeably. We will later show how the maximum estimation error is decreased by increasing S. Clearly, the problem of covering the tuple space with equi-depth buckets does not have a unique solution. For example, Figures 6.1(B) and 6.1(C) show two different solutions with 5 buckets, each bucket having 4 tuples. It clearly seems infeasible to design an algorithm that can come up with ad-hoc solutions, such as seen in Figures 6.1(B) and 6.1(C). Instead, we developed the following algorithm for determining the boundaries of the equi-depth buckets. We will describe the algorithm for the 2-dimensional case. The extension to higher dimensions is straightforward.

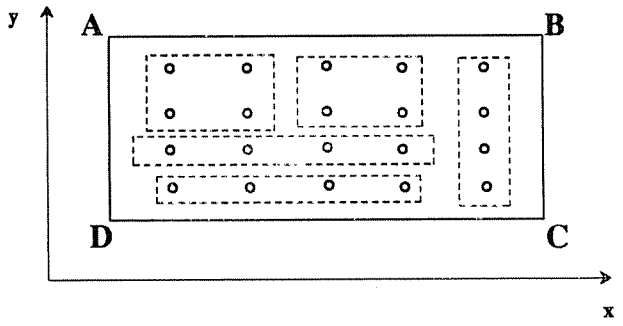
Let the number of buckets desired be $S = bucket_1 * bucket_2$. $Bucket_i$ will be used to denote the number of divisions along the ith attribute (dimension). Thus, the number of tuples in each bucket = $\frac{N}{S}$, where N is the total number of tuples. To simplify the following explanation, we will assume that $\frac{N}{S}$ is an integral number. We assume a sorting routine called SORT which takes three parameters. The first param-




(A)




(B)




(C)

**Figure 6.1.**

eter is the attribute_number (1 or 2 in this case) on which the relation is to be sorted in ascending order. The second and the third parameters are respectively called low and high. Low and high are the serial numbers of two tuples in the relation, such that the tuples ranging from low through high are sorted on the attribute given by the first parameter. For example, the invocation SORT(2, 501, 1000) would sort tuples 501 through 1000 on the second attribute in ascending order. We describe the algorithm in words and then

in pseudo-code.

First, the entire relation (tuples 1 through N) is first sorted on the first attribute. We then form $bucket_1$ partitions of equal size. The first partition consists of tuples 1 through $\frac{N}{bucket_1}$; The second partition consists of tuples ($\frac{N}{bucket_1}$ + 1) through 2 * $\frac{N}{bucket_1}$, etc. We call these partitions **primary** partitions. We then sort each of these primary partitions on the second attribute and then divide each primary partition into $bucket_2$ **secondary** partitions. The important point is that the secondary partitions that are formed from a single primary partition are completely enclosed within that parent primary partition. We thus form a total of ($bucket_1$ * $bucket_2$) number of secondary partitions, each containing $\frac{N}{bucket_1 * bucket_2}$ number of tuples. Each of these secondary partitions corresponds to a bucket and vice versa. Each bucket may be represented by the coordinates of its left-bottom and right-top corners. The left-bottom x(y)-coordinate of a bucket is simply the lowest value of the first (second) attribute of the tuples in the corresponding secondary partition. Similarly, the right-top x(y)-coordinate of a bucket is the highest value of the first (second) attribute of the tuples in the corresponding secondary partition.

We now present the pseudo-code version of the algorithm.

**Algorithm** /* To generate equi-depth 2-dimensional histograms */

SORT (1, 1, N) /* sort the whole relation on the first attribute */

FOR i = 1 TO $bucket_1$ DO
BEGIN

low = (i - 1) * $\frac{N}{bucket_1}$ + 1;

high = i * $\frac{N}{bucket_1}$;

SORT (2, low, high); /* Sort on the second attribute */

END_FOR

capacity = $\frac{N}{bucket_1 * bucket_2}$;

bucket_no = 0;

FOR j = 1 TO $bucket_1$ DO
BEGIN
FOR k = 1 TO $bucket_2$ DO
BEGIN
bucket_no = bucket_no + 1

/* find serial numbers of the first and last tuples in the partition */

first_tuple_id = (bucket_no - 1) * capacity + 1;

last_tuple_id = bucket_no * capacity;

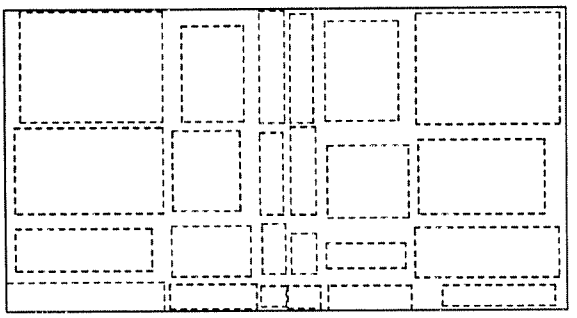
/* Find the coordinates of the lower left and upper right corners of the bucket */

FIND_COORDINATES (bucket_no, first_tuple_id, last_tuple_id);

END_FOR
END_FOR
**End Algorithm**

Figure 6.2 shows an example of 2-dimensional histograms. The values of the first attribute are normally distributed and those along the second attribute have a zipfian [Zipf49] distribution. Equi-depth histograms 'capture' the notion of distribution of the tuples very elegantly. Note that the entire tuple space is not covered by the histograms. This is because there were not any tuples in those spaces not covered by any histogram.

The above algorithm can be easily extended to higher dimensions. For example, extending to three dimensions, we would need to sort each of the ($bucket_1$ * $bucket_2$) secondary partitions on the third attribute and divide each secondary partition into $bucket_3$ **tertiary** partitions. Again, each of the tertiary



**An example of two-dimensional, equi-depth histograms.**

**Figure 6.2.**

partitions are completely enclosed within the parent secondary partition. We would then have a total of ($bucket_1$ * $bucket_2$ * $bucket_3$) number of partitions in a strict hierarchy, each having the same number of tuples. Each of these tertiary partitions corresponds to a 3-dimensional bucket, whose coordinates can be found in the manner described above.

A natural question that arises is: What is the cost of building these histograms? Let the number of dimensions be 3, and the number of data pages in the relation be Z. We will assume that all the tuples in the relation are of the same size. The cost of sorting the whole relation to obtain the $bucket_1$ number of equi-sized primary partitions is

$$C_1 = C * Z * \log(Z)$$

where C is some constant. Each of these primary partitions is sorted to give $bucket_2$ number of secondary partitions. The sorting cost at this stage is therefore given by

$$C_2 = bucket_1 * C * (\frac{Z}{bucket_1}) * \log(\frac{Z}{bucket_1})$$

$$= C * Z * \log(\frac{Z}{bucket_1})$$

Similarly, at the third stage, the sorting cost is given by

$$C_3 = bucket_1 * bucket_2 * C * (\frac{Z}{bucket_1 * bucket_2}) * \log(\frac{Z}{bucket_1 * bucket_2})$$

$$= C * Z * \log(\frac{Z}{bucket_1 * bucket_2})$$

Thus, the total sorting cost = $C_1 + C_2 + C_3$

$$= C * Z \left[ \log(Z) + \log(\frac{Z}{bucket_1}) + \log(\frac{Z}{bucket_1 * bucket_2}) \right]$$

Notice that the sorting cost decreases at each stage. Generalizing to d-dimensions, and assuming the same number of divisions at each stage (= b), the total sorting cost is

$$C * Z * \log \left[ \frac{Z^d}{b^{\frac{d(d-1)}{2}}} \right]$$

At some stage, it is quite possible that the partitions become sufficiently small enough that they can each fit in main memory. If this happens, the sorting cost will be further reduced.

### 6.2. A Storage Structure for Multi-Dimensional Histograms: The H-tree

A multi-dimensional query corresponds to finding all tuples that have attribute values within the bounds of the multi-dimensional box specified by the query (the query box).

**Definition:** An **f-bucket** is a bucket that is completely enclosed within the query box. □

**Definition:** A **p-bucket** is a bucket that partially overlaps the query box. □

Let S be the total number of equi-depth buckets with $\frac{N}{S}$ tuples per bucket. For a given query box, let

f = total number of f-buckets.

p = total number of p-buckets.

Clearly, the following holds:

$$f * \frac{N}{S} \leq \text{actual number of tuples in the query box} \leq (f+p) * \frac{N}{S}.$$

Whatever the method we use to estimate the number of tuples in the query box, we will be interested in determining the exact values of f and p for the given query box. One possible scheme is to check every bucket and see if it is an f-bucket or a p-bucket or neither. This process will obviously become increasingly inefficient for larger values of S, even when the histograms are stored in main memory. In addition, S can grow exponentially with the number of dimensions. We would like a main memory data structure that will enable us to search significantly less than S buckets for f-buckets and p-buckets. At the same

time, the memory requirements for the data structure should grow only linearly with S. Fortunately, the R-tree index structure proposed in [Guttman84] is very close to what is needed. The R-tree mechanism is used to retrieve data items efficiently according to their spatial locations. For our case, the equi-depth buckets correspond to the data items in the leaves of the R-tree. In order to enhance the performance of the search process, we will use a very close variant of the R-tree that exploits the strict hierarchy of partitions obtained during the process of generating the equi-depth buckets. The variant will be called the H-tree (Histogram tree), so as to distinguish it from the R-tree. Unlike the dynamic R-tree, the H-tree will be a static structure that is built once when the histograms are first computed. If the histograms become dated, the H-tree will need to be built again[27]. The H-tree will always be height balanced with the height equal to the number of dimensions. Each level corresponds to the respective dimension. For example, the root node corresponds to the first dimension, the second level to the second dimension and so on.

There are two kinds of nodes in the H-tree:

1. The internal nodes (including the root node), and
2. The leaf nodes.

For ease of notation, we will assume that the data type of the attribute along the kth dimension is DATA_TYPE_k. Let d be the total number of dimensions. An internal node of the H-tree at the kth ($1 \leq k \leq d-1$) level is an array of records and can be characterized by the following definitions:

```
TYPE Internal_node_element_k =

RECORD
{
        low_point, high_point : DATA_TYPE_k;
        next : POINTER;
}
```

```
TYPE Internal_node_k =

ARRAY [1 .. buckets_k] OF Internal_node_element_k;
```

A leaf node is an array of records and can be characterized by the following definitions:

---

[27]We feel that it would be very inefficient to dynamically update the histograms after each addition or deletion.

TYPE **Leaf_node_element =**

```
RECORD
{
        low_coordinate_1, high_coordinate_1 : DATA_TYPE_1;
        low_coordinate_2, high_coordinate_2 : DATA_TYPE_2;
        . . .           . . .
        low_coordinate_d, high_coordinate_d : DATA_TYPE_d;
}
```

TYPE **Leaf_node =**

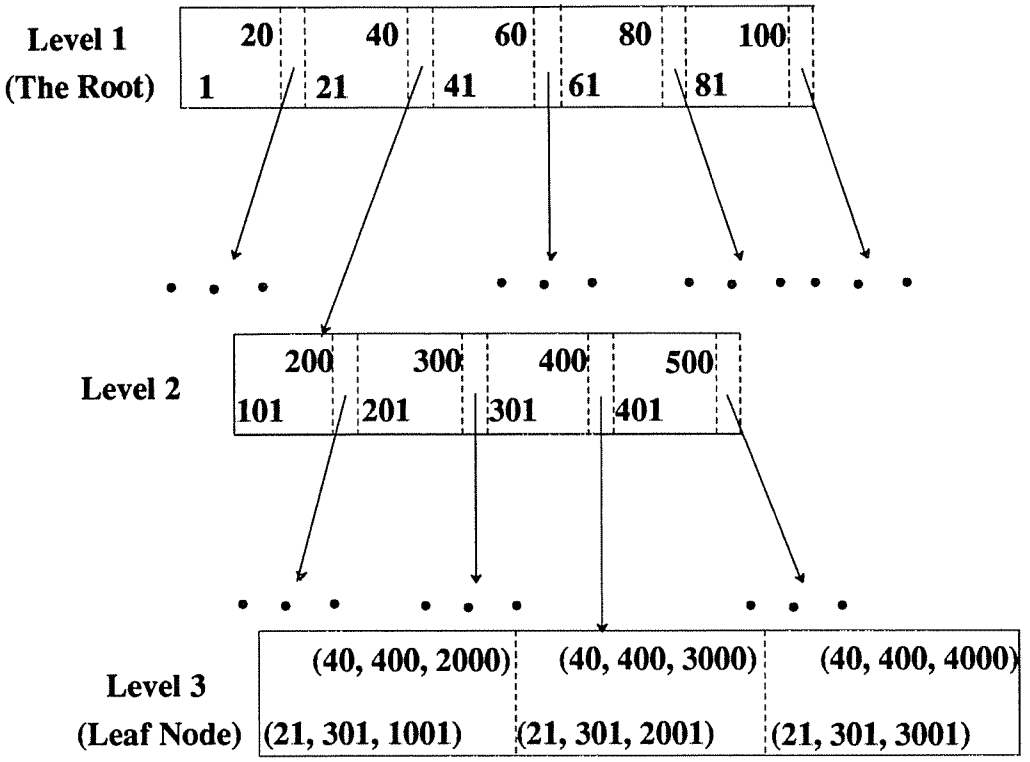
ARRAY [1 .. $buckets_d$] OF Leaf_node_element;

### 6.3. The Search Algorithm

The search algorithm is recursive and similar to that of the search mechanism in the R-tree. We will illustrate it with a three dimensional example. Let the attribute values of the first attribute range from 1 to 100; the attribute values of the second attribute range from 101 to 500; and the attribute values of the third attribute range from 1001 to 4000. Let $bucket_1 = 5$, $bucket_2 = 4$, and $bucket_3 = 3$. Assuming that the attribute values along each dimension are perfectly uniformly distributed, then the resulting H-tree will be as shown in Figure 6.3. The numbers shown in Figure 6.3 represent the values of the fields in the respective records (as defined above).

Let the query box of interest be given by ((31, 325, 1250), (50, 375, 2500)). Walking through the elements in the root, we find that the range (31, 50) overlaps with the second and third element in the root. Following the second pointer, to the second level, we find that (325, 375) overlaps only with the third entry. Following the third pointer into the third level, and searching through the elements at the leaf level, we find that the first and second entries overlap with (1250, 2500). Both these buckets, ((21, 301, 1001), (40, 400, 2000)) and ((21, 301, 2001), (40, 400, 3000)) are p-buckets with respect to the query box. Backing up to the second level and then to the root, we follow the third pointer in the root node down the H-tree in a similar fashion.

It must be observed that, like R-Tree traversals but unlike B-tree traversals, more than one subtree under a node may have to be searched. Hence, in the worst case, the whole H-tree may be traversed. However, in practice, query boxes will generally be small in comparison to the size of the entire tuple space. Only those buckets in the vicinity of the query box will be searched. If the number of entries at each




**A Three dimensional H-Tree.**

**Figure 6.3.**

node is large ( > 20), binary search can be used at each node instead of a linear search.

The storage requirements of the H-tree are dominated by the Leaf nodes. The number of Leaf Node elements is exactly the same as the number of buckets. In addition, for a fixed number of dimensions and a particular set of attributes, the size of a Leaf Node element is fixed. The size of the H-tree thus grows linearly with the number of buckets. Assuming d = 3 and four-byte integer attributes, each leaf_node_element will have six integer fields and thus will be 24 bytes in size. If S = 10 x 10 x 10, the H-tree will occupy slightly over 1000 * 24 = 24,000 bytes of memory.

### 6.4. Estimation Schemes

Consider a relation with N tuples. For a given query box, let 'act_tuples' denote the actual number of tuples within the box. Let 'est_tuples' denote the estimated number of tuples within the box by some

estimation scheme. Consider the two evaluation metrics, D and R, defined as follows: $D = \frac{|\,act\_tuples - est\_tuples\,|}{N}$ and $R = \frac{est\_tuples}{act\_tuples}$, (act_tuples ≠ 0). If the estimation scheme is good, we would expect D to be close to 0, and R to be close to 1. In judging how good an estimation scheme is, we will consider only D for the following reason. Consider the following two scenarios:

1. act_tuples = 10; est_tuples = 100;

2. act_tuples = 1000; est_tuples = 10000;

In either case R = 10. However, $D = \frac{90}{N}$ in the first case and $D = \frac{9000}{N}$ in the second case. D reflects the magnitude of the error in the estimated selectivity of the query. Since cost formulas in query optimizers depend heavily on the estimated selectivity factors, D is a much better metric than R. Assuming that $\frac{90}{N}$ is fairly small, it is unlikely that the access path chosen in the first scenario, based on the estimated selectivity, would be different from the optimal access path. On the other hand, it is quite possible that the access path chosen in the second scenario might be different from the optimal access path, since $\frac{9000}{N}$ is 100 times larger than $\frac{90}{N}$. Therefore, we will use only parameter D for judging the quality of an estimation scheme.

We now describe two schemes for estimating the value of est_tuples for a given query box. The first scheme, viz., the **Half Scheme**, is conservative in that it only attempts to reduce the worst case error. The second scheme, viz., the **Uniform Scheme**, as we will demonstrate below, performs much better on the average. Theoretically, the worst case error possible using the Uniform Scheme is twice that of the Half Scheme. However, in practice, we have found that the maximum error of the Uniform Scheme is significantly less than the maximum error attained by the Half Scheme. After describing the two schemes, we will present expermental results that confirm these statements.

### 6.4.1. The Half Scheme

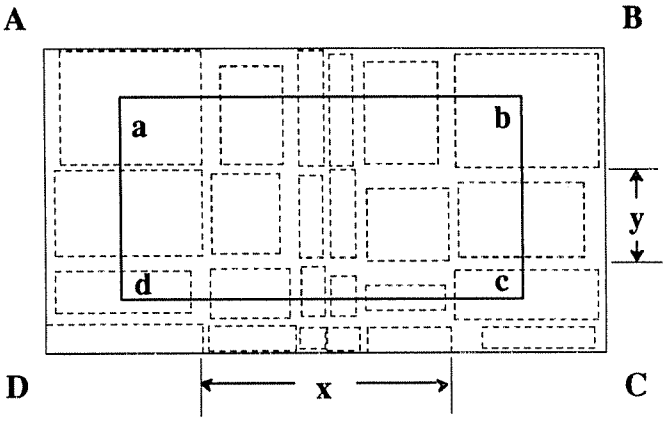
Given a query box, we know that the following holds:

$$f * \frac{N}{S} \leq \text{actual number of tuples in the query box} \leq (f+p) * \frac{N}{S},$$

where f(p) is the number of f(p)-buckets for the given query box. In other words,

$$\frac{f}{S} \leq \text{actual selectivity} \leq \frac{(f+p)}{S}.$$

If we choose the estimated selectivity to be $\frac{(f + \frac{p}{2})}{S}$, which is the mid-point of the two extremes, our estimation error can never be larger than $\frac{p}{2S}$. In other words, for every partially overlapping bucket, we will assume that half of the tuples within it are also within the given query box. How large can p get? Figure 6.4 shows an example of 2-dimensional (d = 2) histograms. Clearly[28], f = x * y and p = (x + 2) * (y + 2) - x * y = 2 * (x + y) + 4. Note that the estimation error for small query boxes will be smaller. p will assume its largest value when x is $bucket_1$ - 2, and y is $bucket_2$ - 2. The largest value of p is therefore 2 * ($bucket_1$ + $bucket_2$) - 4. The largest estimation error is thus = $\frac{p}{2S} = \frac{bucket_1 + bucket_2 - 2}{bucket_1 * bucket_2} \approx \frac{1}{bucket_1} + \frac{1}{bucket_2}$. If $bucket_1 = bucket_2 = 5$, the maximum estimation error = 32%. Similarly we




**Rectangle ABCD is the entire tuple space.**

**Rectangle abcd is a query box.**

**x = 4, y = 1.**

**bucket(1) = 6, bucket(2) = 4.**

**Figure 6.4.**

[28]A similar analysis can be carried out for any value of d. In particular, for d = 3, we have p = (x + 2) * (y + 2) * (z + 2) - x * y * z.

can show that the largest estimation error in d dimensions

$$\approx \sum_{i=1}^{d} \frac{1}{\text{bucket}_i}$$

Thus, for a fixed number of buckets S, we can easily show (by differentiating the expression above and equating it to zero and solving for $\text{bucket}_i$, i = 1, d) that the maximum estimation error is minimized when

$$\text{bucket}_i = S^{\frac{1}{d}}, \; i = 1, d.$$

### 6.4.2. The Uniform Scheme

In this scheme, the estimated number of tuples for a given query box is calculated by the following formula:

$$\text{est\_tuples} = \frac{N}{S}(f + \sum_{i=1}^{p} \text{fract}(i))$$

where

$$\text{fract}(i) = \frac{\text{Size of ((ith p–bucket)}\cap\text{(the query box))}}{\text{(Size of the ith p–bucket)}}$$

where if d = 2, the size refers to the area; if d = 3, the size refers to the volume. Clearly,

$$f * \frac{N}{S} < \text{est\_tuples} < (f + p) * \frac{N}{S}, \text{ since } 0 < \text{fract}(i) < 1 \text{ for } i = 1, p.$$

Essentially, we are assuming that tuples are uniformly distributed in each of the p-buckets. As might be expected, the validity of this assumption will be enhanced as the size of each bucket becomes smaller, regardless of the actual distribution of the tuples. As we will demonstrate below, our experimental results bear this out. It is possible that the error in the estimate can be as large as $\frac{p}{S}$. However, this never occurred in our experiments.

## 6.5. The Experiments

Each relation had 104,000 tuples. All attributes were integers with attribute values varying from a minimum of 1 to a maximum[29] of 241. The values of each attribute were generated independently of each other and had one of the following three distributions: normal(n) (with a mean of 121 and a standard

[29]These bounds were chosen arbitrarily. In fact, we replaced the bound 241 by 10001 and repeated some of the experiments. There was no significant change in the results.

deviation of 50), uniform(u), or zipfian(z). We generated a total of 9 (= 3 * 3) tuple spaces in the 2-dimensional experiments and a total of 27 (= 3 * 3 * 3) in the 3-dimensional experiments. For each tuple distribution, we performed two types of experiments.

The objective of the first series of experiments was to observe the maximum estimation error obtained by the two schemes. A total of 5000 large, square query boxes were generated such that they almost occupied the entire tuple space. Using these large boxes, we measured the maximum estimation error for both the estimation schemes. The results of the first series of experiments (on two dimensions) are presented in Table 6.1. The corresponding 3-dimensional results are presented in Section 6.7. Most of the tables presented here have a format similar to that of Table 6.1. The first column indicates the distributions of the attributes along each of the dimensions. For example, an entry "n z" indicates a normal

| distr | stats | 5x5 | | 10x10 | | 20x20 | |
|---|---|---|---|---|---|---|---|
| | | Half | Uniform | Half | Uniform | Half | Uniform |
| n n | max | **28.3+** | **16.1+** | **14.3+** | **6.6+** | 6.5- | **2.2-** |
| | avg | **18.3** | **8.1** | **7.3** | **2.8** | **2.3** | **0.6** |
| | std.dev | 7.5 | 3.9 | 4.2 | 1.5 | 1.6 | 0.5 |
| n u | max | 27.6+ | 6.6+ | 13.6+ | 3.0+ | 5.8+ | 1.2- |
| | avg | 12.1 | 3.4 | 4.9 | 1.2 | 1.7 | 0.4 |
| | std.dev | 7.4 | 1.8 | 3.4 | 0.8 | 1.2 | 0.3 |
| n z | max | 23.6+ | 11.1+ | 11.0+ | 3.8+ | 5.1+ | 1.6+ |
| | avg | 8.8 | 1.6 | 4.3 | 0.6 | 1.7 | 0.2 |
| | std.dev | 5.9 | 1.5 | 2.8 | 0.6 | 1.1 | 0.2 |
| u n | max | 27.7+ | 6.6+ | 13.8+ | 3.0+ | 6.4+ | 1.2- |
| | avg | 12.2 | 3.5 | 5.1 | 1.2 | 1.9 | 0.3 |
| | std.dev | 7.4 | 1.7 | 3.4 | 0.8 | 1.3 | 0.2 |
| u u | max | 28.2- | 0.7- | 13.2+ | 0.6+ | **7.1-** | 0.5- |
| | avg | 9.0 | 0.2 | 3.8 | 0.1 | 1.5 | 0.1 |
| | std.dev | 6.9 | 0.1 | 2.9 | 0.1 | 1.2 | 0.1 |
| u z | max | 22.7+ | 6.1- | 10.6+ | 1.5+ | 5.3+ | 1.3+ |
| | avg | 6.4 | 1.1 | 3.1 | 0.3 | 1.3 | 0.2 |
| | std.dev | 5.0 | 0.9 | 2.3 | 0.2 | 0.9 | 0.2 |
| z n | max | 23.6+ | 11.0+ | 12.3+ | 4.6+ | 5.7- | 1.4- |
| | avg | 9.0 | 1.7 | 4.6 | 0.8 | 1.8 | 0.3 |
| | std.dev | 5.9 | 1.6 | 2.9 | 0.7 | 1.2 | 0.2 |
| z u | max | 22.6+ | 6.1- | 11.8+ | 2.0+ | 4.8+ | 0.9+ |
| | avg | 6.5 | 1.1 | 3.3 | 0.4 | 1.4 | 0.2 |
| | std.dev | 5.1 | 1.0 | 2.4 | 0.4 | 1.0 | 0.2 |
| z z | max | 21.1+ | 9.1- | 9.7+ | 2.2+ | 4.1+ | 1.0+ |
| | avg | 4.0 | 1.0 | 2.4 | 0.3 | 1.1 | 0.1 |
| | std.dev | 3.7 | 1.0 | 2.0 | 0.3 | 0.8 | 0.2 |

**Table 6.1: Estimation Errors For Large Boxes By The Two Schemes.**

distribution for the first attribute and a zipfian distribution for the second attribute. For a given tuple distribution, we varied the number of equi-depth buckets from 25 (5x5) to 100 (10x10) and finally to 400 (20x20). In each case, we calculated the actual number of tuples and the estimated number of tuples within a query box by each of the two schemes. This was repeated for each of the 5000 large query boxes. For each scheme, the magnitude of the maximum percentage deviation in selectivity over these 5000 boxes was calculated. A positive (negative) sign besides each number in the "max" row indicates that the actual number of tuples was greater than or equal to (less than) the estimated number of tuples. We also present the average ("avg" row) and the standard deviation ("std.dev" row) of the percentage magnitudes of the deviations. The largest maximum deviation and the largest average deviation values for each column are indicated in boldface.

There are some obvious conclusions we can draw from Table 6.1. We know from Section 6.4.1 that the maximum percentage error in estimating the selectivity by the Half scheme is 32% for the 5x5 case. The corresponding numbers for the 10x10 and the 20x20 case are 18% and 9.5% respectively. The maximum estimation error in each column (in Table 6.1) under the Half Scheme are close to the theoretical limits. On the other hand, the corresponding maximum estimation errors obtained by the Uniform Scheme are about one-half to one-third of those obtained by the Half Scheme.

The objective of the second series of experiments was to study the average behavior of the Uniform Scheme. A total of 5000 square query boxes were generated such that a large percentage of boxes had small areas (or volumes). This reflects real life situations wherein a large percentage of queries retrieve only a small amount of data. The coordinates of the query boxes were chosen from a uniform distribution. Table 6.2 gives the distribution of the areas of the boxes. The number of tuples in each query box was estimated by the Uniform Scheme only. The results of these experiments are displayed in Table 6.3. We generated histograms of equal depth as well as histograms of equal width[30]. When buckets are of equal width, each bucket has a different number of tuples. When using equi-width histograms and the Uniform Scheme of estimation, the number of tuples within a query box is calculated by the following formula:

[30]The width of each histogram along a specific attribute is the same. Thus in two dimensions, the buckets have the same area.

| range of areas | Number of boxes |
|---|---|
| 0 - 2499 | 1069 |
| 2500 - 4999 | 446 |
| 5000 - 7499 | 327 |
| 7500 - 9999 | 265 |
| 10000 - 12499 | 232 |
| 12500 - 14999 | 214 |
| 15000 - 17499 | 189 |
| 17500 - 19999 | 182 |
| 20000 - 22499 | 147 |
| 22500 - 24999 | 186 |
| 25000 - 27499 | 161 |
| 27500 - 29999 | 175 |
| 30000 - 32499 | 148 |
| 32500 - 34999 | 168 |
| 35000 - 37499 | 137 |
| 37500 - 39999 | 132 |
| 40000 - 42499 | 158 |
| 42500 - 44999 | 138 |
| 45000 - 47499 | 98 |
| 47500 - 49999 | 126 |
| 50000 - 52499 | 138 |
| 52500 - 54999 | 100 |
| 55000 - 57499 | 97 |
| 57500 - 59999 | 27 |

**Table 6.2.**
**Distribution Of The Areas Of Query Boxes.**

| distr | stats | 5x5 | | 10x10 | | 20x20 | |
|---|---|---|---|---|---|---|---|
| | | ED | EW | ED | EW | ED | EW |
| n n | max | **15.9+** | 5.5- | **6.1+** | 9.9- | **1.4-** | 18.4- |
| | avg | **3.2** | 1.2 | **1.0** | 3.4 | **0.3** | **7.4** |
| | std.dev | 3.3 | 1.1 | 1.1 | 3.0 | 0.3 | 6.3 |
| n u | max | 6.6+ | 4.6- | 2.8+ | 9.0- | 1.2- | 17.6- |
| | avg | 1.5 | 1.1 | 0.5 | 3.1 | 0.2 | 6.5 |
| | std.dev | 1.6 | 1.1 | 0.6 | 2.7 | 0.2 | 5.6 |
| n z | max | 11.2+ | 35.4- | 3.2+ | 31.3- | 1.3+ | 30.6- |
| | avg | 0.8 | 6.6 | 0.2 | 3.9 | 0.1 | 3.6 |
| | std.dev | 0.8 | 10.8 | 0.3 | 7.5 | 0.1 | 5.8 |
| u n | max | 6.5+ | 4.5- | 2.7+ | 8.9- | 1.1- | 17.5- |
| | avg | 1.5 | 1.0 | 0.5 | 3.0 | 0.1 | 6.4 |
| | std.dev | 1.5 | 1.1 | 0.6 | 2.6 | 0.1 | 5.6 |
| u u | max | 0.6- | 3.3- | 0.5+ | 7.5- | 0.3- | 16.4- |
| | avg | 0.1 | 1.2 | 0.1 | 2.7 | 0.1 | 5.7 |
| | std.dev | 0.1 | 1.1 | 0.1 | 2.4 | 0.1 | 5.0 |
| u z | max | 6.5- | 33.9- | 1.5- | 29.7- | 1.2- | 29.8- |
| | avg | 0.8 | 6.2 | 0.2 | 3.5 | 0.1 | 3.3 |
| | std.dev | 0.8 | 10.2 | 0.2 | 7.0 | 0.1 | 5.5 |
| z n | max | 8.5+ | 35.4- | 3.7+ | 31.2- | 1.4- | 30.7- |
| | avg | 0.8 | 6.7 | 0.2 | **3.9** | 0.1 | 3.8 |
| | std.dev | 0.8 | 10.8 | 0.3 | 7.5 | 0.1 | 5.9 |
| z u | max | 6.5- | 34.0- | 2.0+ | 29.8- | 0.9+ | 30.0- |
| | avg | 0.9 | 6.3 | 0.2 | 3.6 | 0.1 | 3.4 |
| | std.dev | 0.8 | 10.2 | 0.3 | 7.1 | 0.1 | 5.6 |
| z z | max | 11.3- | **48.6-** | 2.6- | **41.3-** | 1.0- | **37.2-** |
| | avg | 0.6 | **6.8** | 0.2 | 3.4 | 0.1 | 2.2 |
| | std.dev | 0.9 | 12.9 | 0.2 | 8.4 | 0.1 | 5.7 |

**Table 6.3: Estimation Errors For Zipfian Boxes By The Uniform Scheme.**

$$\text{est\_tuples} = \sum_{i=1}^{f+p} (\text{fract(i)}*\text{occupancy(i)})$$

where occupancy(i) is the number of tuples in the ith bucket. The one advantage of building equi-width buckets is that the relation never has to be sorted. However, as Table 6.3 shows, the maximum deviations obtained by the equal-width scheme (shown under the columns titled 'EW') are very high compared to the maximum deviations obtained by the equal-depth scheme (shown under the columns titled 'ED'). In the 20x20 case, the largest average deviation is only 0.25% for equi-depth histograms as opposed to 7.38% for equi-width histograms.

A natural question that arises at this point is the following: are the results presented in Table 6.3, especially those under the Equi-depth columns, statistically valid? To find out, we used the **method of batch means** [Sargent76]. We generated 20 batches consisting of 1000 query boxes each such that the

sizes of the boxes in each batch formed a zipfian distribution. The boxes were located randomly. In each batch we estimated the number of tuples in each query box using the Uniform Scheme. For each of these batches, we calculated the average percentage deviation (as before) for each tuple distribution and each equi-depth histogram configuration. In every case, we found that with 90% confidence, the variance of the average was less than 8% of the average. In fact, out of a total of 27 cases, the variance of the average was less than 6% of the average in 23 of the cases. In all cases, the variance itself did not exceed 0.152%.

### 6.6. Building Histograms by Random Sampling

In situations where sorting a relation may be considered expensive or where only a quick estimate of the selectivity is required, we can resort to building equi-depth histograms using a small sample of tuples taken from the base relation. We adopted the **random sampling technique without replacement** [Gibbons76] to obtain our sample. A random sample satisfies the property that, for a finite population and a fixed sample size n, every element in the population has the same chance of being included in the sample and every combination of of n elements has an equal chance of being the sample selected. During the sampling process, the same tuple is not picked more than once. The usefulness of sampling was explored in [Shapiro84] for building 1-dimensional histograms. In what follows, we will show that sampling is also very beneficial for building multi-dimensional histograms at a very low cost.

#### 6.6.1. The Kolmogorov Statistic

Let $\alpha$ be the proportion of tuples in the population (relation) that satisfy a certain property. In our case, this property is that they lie within a certain query box. Let $\beta$ be the proportion of tuples in the sample that lie within the same query box. Then the Kolmogorov's statistic [Gibbons76] tells us that $|\alpha - \beta| \leq d$ with probability $\geq p$ if the sample size is at least n. d is called the precision and p is the confidence. Given the values of p and d, n can be found using standard tables. One such table is reproduced here from [Gibbons76], page 73.

Thus, when the sample size is chosen to be 1063, we can say that $|\alpha - \beta| \leq 0.05$ with confidence $\geq 0.99$. For a fixed confidence, the sample size is inversely proportional to the square of the precision.

| p/d | 0.80 | 0.90 | 0.95 | 0.98 | 0.99 |
|---|---|---|---|---|---|
| 0.05 | 458 | 596 | 740 | 937 | 1063 |
| 0.10 | 115 | 149 | 185 | 231 | 266 |
| 0.15 | 51 | 67 | 83 | 105 | 119 |

**Minimum sample size required to estimate with precision d and confidence p.**

In our experiments[31], we chose our sample size to be 1200. Thus with confidence[32] 0.99, we can say that $|\alpha - \beta| \leq 0.0471$. We present the results of our experiments in Table 6.4. The 5000 query boxes used for this test were the same as those used in generating Table 6.3. The numbers in the first three columns

| distr | stats | 5x5 | 10x10 | 20x20 | Sample |
|---|---|---|---|---|---|
| | | ED | ED | ED | |
| n n | max | **15.9+** | **5.5+** | **4.0+** | **4.1+** |
| | avg | **3.1** | **1.0** | 0.7 | 0.7 |
| | std.dev | 3.3 | 0.9 | 0.6 | 0.6 |
| n u | max | 8.0+ | 3.9+ | 3.0+ | 2.6+ |
| | avg | 1.7 | 0.6 | 0.5 | 0.4 |
| | std.dev | 1.7 | 0.5 | 0.4 | 0.4 |
| n z | max | 11.6+ | 4.5+ | 3.4+ | 3.3+ |
| | avg | 0.7 | 0.6 | 0.5 | 0.6 |
| | std.dev | 0.7 | 0.5 | 0.4 | 0.5 |
| u n | max | 6.5+ | 3.2+ | 3.7- | 3.3- |
| | avg | 1.5 | 0.9 | 0.9 | 0.9 |
| | std.dev | 1.2 | 0.8 | 0.8 | 0.8 |
| u u | max | 2.3- | 3.9- | 3.8- | 4.1- |
| | avg | 0.7 | 1.0 | **1.1** | **1.0** |
| | std.dev | 0.7 | 0.9 | 0.9 | 0.9 |
| u z | max | 4.6- | 2.4- | 2.5+ | 2.6+ |
| | avg | 0.6 | 0.3 | 0.5 | 0.5 |
| | std.dev | 0.6 | 0.3 | 0.4 | 0.5 |
| z n | max | 8.6+ | 3.3+ | 2.3+ | 2.3+ |
| | avg | 0.7 | 0.3 | 0.4 | 0.4 |
| | std.dev | 0.8 | 0.3 | 0.3 | 0.3 |
| z u | max | 6.4- | 2.9- | 3.0+ | 2.8+ |
| | avg | 1.2 | 0.8 | 0.7 | 0.7 |
| | std.dev | 1.1 | 0.6 | 0.6 | 0.5 |
| z z | max | 8.7- | 2.3+ | 2.4+ | 2.4+ |
| | avg | 0.7 | 0.3 | 0.2 | 0.2 |
| | std.dev | 0.8 | 0.3 | 0.3 | 0.3 |

**Table 6.4: Estimation Errors By Sampling With The Uniform Scheme.**

[31] 1200 is the smallest multiple of 400 (20 * 20) larger than 1063.

[32] $0.0471 \approx (1063 * (0.05)^2/1200)^{0.5}$.

were obtained by building equal-depth histograms using the tuples obtained in the sample. In the last column (titled 'sample'), the estimated number of tuples within a query box was calculated using the following formula:

$$est\_tuples = \beta * N,$$

where N is the number of tuples in the whole relation and $\beta$ was the actual fraction of tuples in the sample that was within the query box. In other words, we assumed that the fraction of tuples in the sample (that was within the query box) was the same as the fraction of tuples in the entire population (that was within the same query box).

Comparing the equi-depth columns of Table 6.3 with the corresponding ones in Table 6.4, we see that the sampling technique performs very well. We calculated the differences between the corresponding "avg" values in Table 6.3 and Table 6.4. The maximum difference between the corresponding "avg" entries were as follows: 0.588% (5x5), 0.892% (10x10), and 1.005% (20x20). Thus the estimates obtained from the histograms built using the sample are well within the tolerance expected (4.71%).

### 6.7. The Three Dimensional Results

We conducted the same series of experiments on three dimensions as those on two dimensions. The objective of the first series of experiments, as before, was to observe the maximum estimation error obtained by the two schemes. We generated 5000 cubic boxes large enough to maximize the number of p-buckets. The results are presented in the three parts of Table 6.5. The first column indicates the distributions of the attributes along the three dimensions. For a given tuple distribution, we varied the number of equi-depth buckets from 125 (5x5x5) to 1000 (10x10x10) and finally to 8000 (20x20x20). As demonstrated in Section 6.4.1, we can easily calculate that the maximum percentage error in estimating the selectivity by the Half Scheme for each of the above bucket configurations. For the 5x5x5 case, the maximum percentage error in estimating the selectivity by the Half Scheme is 39.2%. The corresponding numbers for the 10x10x10 and 20x20x20 case are 24.4% and 13.55% respectively. The maximum estimation error in each column of Table 6.5 under the Half scheme is again close to the theoretical limits for many tuple distributions. On the other hand, both the maximum and the average estimation errors obtained by the Uniform Scheme are also significantly smaller than those obtained by Half Scheme. Again, we conclude that the Uniform Scheme performs better than the Half Scheme.

| distr | stats | 5x5x5 | | 10x10x10 | | 20x20x20 | |
|---|---|---|---|---|---|---|---|
| | | Half | Uniform | Half | Uniform | Half | Uniform |
| n n n | max | 34.2+ | **20.4+** | 19.4+ | **8.9+** | 7.4+ | **4.2-** |
| | avg | **22.2** | **9.6** | **10.1** | **3.6** | **2.9** | **1.1** |
| | std.dev | 9.4 | 5.3 | 5.7 | 2.0 | 2.1 | 0.8 |
| n n u | max | 35.0+ | 11.3+ | 20.1+ | 5.0+ | 7.4+ | 3.1- |
| | avg | 16.7 | 5.6 | 7.6 | 2.1 | 2.2 | 0.7 |
| | std.dev | 9.7 | 3.0 | 5.1 | 1.2 | 1.8 | 0.6 |
| n n z | max | 32.1+ | 15.2+ | 16.9+ | 6.5+ | 6.1+ | 3.0- |
| | avg | 10.2 | 2.7 | 5.5 | 1.1 | 1.7 | 0.5 |
| | std.dev | 7.2 | 2.1 | 3.8 | 0.8 | 1.3 | 0.5 |
| n u n | max | 35.1+ | 11.3+ | 20.2+ | 5.0+ | 7.3+ | 3.1- |
| | avg | 16.8 | 5.6 | 7.6 | 2.1 | 2.2 | 0.7 |
| | std.dev | 9.7 | 3.0 | 5.1 | 1.2 | 1.7 | 0.6 |
| n u u | max | 35.9+ | 4.8+ | 21.1+ | 2.2+ | 7.4+ | 1.9- |
| | avg | 13.1 | 2.5 | 5.7 | 0.9 | 1.6 | 0.4 |
| | std.dev | 9.8 | 1.4 | 4.8 | 0.6 | 1.5 | 0.4 |
| n u z | max | 33.0+ | 7.6+ | 17.3+ | 3.1+ | 5.9+ | 1.8- |
| | avg | 8.1 | 1.4 | 4.1 | 0.6 | 1.2 | 0.3 |
| | std.dev | 6.8 | 1.4 | 3.4 | 0.5 | 1.1 | 0.3 |
| n z n | max | 32.2+ | 16.1+ | 16.7+ | 6.4+ | 6.1+ | 2.9- |
| | avg | 10.7 | 2.8 | 5.7 | 1.2 | 1.8 | 0.5 |
| | std.dev | 7.3 | 2.3 | 4.0 | 0.9 | 1.4 | 0.5 |
| n z u | max | 33.0+ | 7.6+ | 17.4+ | 3.0+ | 5.8+ | 1.8- |
| | avg | 8.4 | 1.4 | 4.4 | 0.6 | 1.3 | 0.3 |
| | std.dev | 7.0 | 1.4 | 3.6 | 0.5 | 1.1 | 0.4 |
| n z z | max | 28.8+ | 11.9- | 14.2+ | 3.7+ | 4.8+ | 1.6- |
| | avg | 4.9 | 1.2 | 2.9 | 0.4 | 0.9 | 0.2 |
| | std.dev | 4.9 | 1.6 | 2.6 | 0.5 | 0.8 | 0.3 |

**Part 1 of Table 6.5: Estimation Errors For Large Boxes By The Two Schemes.**

| distr | stats | 5x5x5 | | 10x10x10 | | 20x20x20 | |
|---|---|---|---|---|---|---|---|
| | | Half | Uniform | Half | Uniform | Half | Uniform |
| u n n | max | 35.1+ | 11.3+ | 20.3+ | 5.0+ | **7.5+** | 3.1- |
| | avg | 16.9 | 5.6 | 7.8 | 2.1 | 2.3 | 0.7 |
| | std.dev | 9.7 | 3.0 | 5.1 | 1.2 | 1.8 | 0.6 |
| u n u | max | 35.8+ | 4.9+ | 20.9+ | 2.2+ | 7.3+ | 1.9- |
| | avg | 13.1 | 2.5 | 5.7 | 0.9 | 1.7 | 0.4 |
| | std.dev | 9.7 | 1.3 | 4.8 | 0.5 | 1.5 | 0.4 |
| u n z | max | 33.0+ | 7.5+ | 17.6+ | 3.0+ | 5.8+ | 1.8- |
| | avg | 8.2 | 1.3 | 4.2 | 0.5 | 1.2 | 0.3 |
| | std.dev | 6.9 | 1.4 | 3.5 | 0.5 | 1.1 | 0.3 |
| u u n | max | 35.9+ | 4.9+ | 21.0+ | 2.2+ | 7.4+ | 1.9- |
| | avg | 13.2 | 2.4 | 5.9 | 0.9 | 1.7 | 0.4 |
| | std.dev | 9.7 | 1.3 | 4.8 | 0.6 | 1.5 | 0.4 |
| u u u | max | **36.7+** | 0.7- | **21.5+** | 0.5- | 7.1+ | 0.8- |
| | avg | 11.9 | 0.3 | 4.9 | 0.2 | 1.4 | 0.2 |
| | std.dev | 9.2 | 0.2 | 4.5 | 0.1 | 1.4 | 0.1 |
| u u z | max | 33.8+ | 6.5- | 17.6+ | 1.5- | 6.0+ | 0.7- |
| | avg | 6.9 | 1.3 | 3.1 | 0.3 | 0.9 | 0.2 |
| | std.dev | 6.5 | 1.2 | 3.2 | 0.3 | 0.9 | 0.1 |
| u z n | max | 33.0+ | 7.6+ | 17.3+ | 3.0+ | 5.7+ | 1.7- |
| | avg | 8.5 | 1.4 | 4.4 | 0.6 | 1.3 | 0.3 |
| | std.dev | 7.0 | 1.4 | 3.6 | 0.5 | 1.1 | 0.3 |
| u z u | max | 33.8+ | 6.5- | 18.2+ | 1.5- | 5.7+ | 1.1- |
| | avg | 7.2 | 1.2 | 3.5 | 0.3 | 1.0 | 0.2 |
| | std.dev | 6.7 | 1.2 | 3.3 | 0.3 | 0.9 | 0.2 |
| u z z | max | 29.7+ | 11.3- | 13.9+ | 2.5- | 4.5+ | 1.0- |
| | avg | 4.2 | 1.4 | 2.2 | 0.3 | 0.7 | 0.2 |
| | std.dev | 4.6 | 1.6 | 2.3 | 0.4 | 0.7 | 0.2 |

**Part 2 of Table 6.5: Estimation Errors For Large Boxes By The Two Schemes.**

| distr | stats | 5x5x5 | | 10x10x10 | | 20x20x20 | |
|---|---|---|---|---|---|---|---|
| | | Half | Uniform | Half | Uniform | Half | Uniform |
| z n n | max | 32.1+ | 15.4+ | 17.3+ | 5.8+ | 5.5+ | 2.9- |
| | avg | 10.7 | 2.8 | 5.8 | 1.1 | 1.8 | 0.5 |
| | std.dev | 7.3 | 2.2 | 4.0 | 0.9 | 1.3 | 0.5 |
| z n u | max | 32.9+ | 7.8+ | 17.6+ | 3.2- | 5.5+ | 1.7- |
| | avg | 8.5 | 1.4 | 4.4 | 0.6 | 1.4 | 0.3 |
| | std.dev | 7.0 | 1.4 | 3.6 | 0.5 | 1.1 | 0.3 |
| z n z | max | 28.8+ | 11.9- | 14.1+ | 3.8- | 4.2+ | 1.6- |
| | avg | 5.0 | 1.2 | 2.9 | 0.4 | 0.9 | 0.2 |
| | std.dev | 5.1 | 1.6 | 2.6 | 0.5 | 0.8 | 0.3 |
| z u n | max | 33.0+ | 7.7+ | 17.6+ | 3.0- | 5.4+ | 1.7- |
| | avg | 8.5 | 1.4 | 4.4 | 0.6 | 1.4 | 0.3 |
| | std.dev | 7.0 | 1.4 | 3.6 | 0.5 | 1.0 | 0.3 |
| z u u | max | 33.8+ | 6.5- | 18.4+ | 1.9- | 5.4+ | 0.6- |
| | avg | 7.3 | 1.1 | 3.4 | 0.4 | 1.0 | 0.2 |
| | std.dev | 6.7 | 1.1 | 3.3 | 0.4 | 0.9 | 0.1 |
| z u z | max | 29.7+ | 11.2- | 13.8+ | 2.4- | 4.0+ | 0.7- |
| | avg | 4.3 | 1.2 | 2.2 | 0.3 | 0.7 | 0.2 |
| | std.dev | 4.7 | 1.5 | 2.3 | 0.4 | 0.6 | 0.1 |
| z z n | max | 28.9+ | 12.0- | 14.2+ | 3.7- | 4.3+ | 1.5- |
| | avg | 5.1 | 1.2 | 3.1 | 0.4 | 1.0 | 0.2 |
| | std.dev | 5.0 | 1.6 | 2.7 | 0.5 | 0.8 | 0.3 |
| z z u | max | 29.7+ | 11.3- | 14.7+ | 2.6- | 4.0+ | 0.9- |
| | avg | 4.4 | 1.2 | 2.5 | 0.3 | 0.7 | 0.2 |
| | std.dev | 4.7 | 1.5 | 2.5 | 0.4 | 0.7 | 0.1 |
| z z z | max | 24.0+ | 14.7- | 11.2+ | 3.0- | 3.2+ | 0.9- |
| | avg | 2.7 | 1.2 | 1.6 | 0.2 | 0.5 | 0.1 |
| | std.dev | 3.3 | 1.8 | 1.7 | 0.4 | 0.5 | 0.1 |

**Part 3 of Table 6.5: Estimation Errors For Large Boxes By The Two Schemes.**

The objective of the second series of experiments was to observe the average behavior of the Uniform Scheme. As before, we generated a total of 5000 cubic query boxes such that a large percentage of them had small volumes. Table 6.6 gives the distribution of the volumes of the boxes. We again generated histograms of equal depth as well as histograms of equal width. We estimated the number of tuples in each query box by the Uniform Scheme and the results are presented in the three parts of Table 6.7.

From Table 6.7, we can see that the maximum deviations obtained by the equi-width scheme are very high. In the 20x20x20 case, the maximum average deviation is only 0.299% when estimating with equi-depth buckets. On the other hand, the maximum average deviation in the 20x20x20 case is over 9% when estimating with equi-width buckets. Notice that the maximum deviation is around 50% in the "z z z" row under all the equi-width columns.

| range of volumes | Number of boxes |
|---|---|
| 0 - 499999 | 1623 |
| 500000 - 999999 | 451 |
| 1000000 - 1499999 | 328 |
| 1500000 - 1999999 | 230 |
| 2000000 - 2499999 | 200 |
| 2500000 - 2999999 | 184 |
| 3000000 - 3499999 | 135 |
| 3500000 - 3999999 | 144 |
| 4000000 - 4499999 | 146 |
| 4500000 - 4999999 | 117 |
| 5000000 - 5499999 | 121 |
| 5500000 - 5999999 | 100 |
| 6000000 - 6499999 | 93 |
| 6500000 - 6999999 | 104 |
| 7000000 - 7499999 | 75 |
| 7500000 - 7999999 | 88 |
| 8000000 - 8499999 | 108 |
| 8500000 - 8999999 | 77 |
| 9000000 - 9499999 | 59 |
| 9500000 - 9999999 | 91 |
| 10000000 - 10499999 | 57 |
| 10500000 - 10999999 | 98 |
| 11000000 - 11499999 | 47 |
| 11500000 - 11999999 | 68 |
| 12000000 - 12499999 | 81 |
| 12500000 - 12999999 | 63 |
| 13000000 - 13499999 | 60 |
| 13500000 - 13999999 | 52 |

**Table 6.6.**
**Distribution Of The Volumes Of Query Boxes.**

| distr | stats | 5x5x5 | | 10x10x10 | | 20x20x20 | |
|---|---|---|---|---|---|---|---|
| | | ED | EW | ED | EW | ED | EW |
| n n n | max | **19.2+** | 9.0- | **8.2+** | 15.8- | **3.2-** | 29.4- |
| | avg | **3.6** | 1.3 | **1.2** | **4.2** | 0.3 | **9.3** |
| | std.dev | 4.4 | 1.7 | 1.6 | 4.5 | 0.4 | 9.6 |
| n n u | max | 10.7+ | 8.2- | 4.7+ | 14.9- | 2.8- | 28.4- |
| | avg | 2.1 | 1.2 | 0.7 | 3.8 | 0.2 | 8.2 |
| | std.dev | 2.6 | 1.7 | 0.9 | 4.1 | 0.3 | 8.7 |
| n n z | max | 10.1+ | 36.7- | 4.1+ | 34.6- | 2.0- | 40.2- |
| | avg | 0.6 | 6.0 | 0.2 | 4.1 | 0.1 | 4.6 |
| | std.dev | 0.9 | 10.7 | 0.3 | 8.2 | 0.1 | 7.7 |
| n u n | max | 11.3+ | 7.3- | 4.8+ | 14.0- | 2.1- | 27.4- |
| | avg | 2.1 | 1.2 | 0.7 | 3.8 | 0.2 | 8.2 |
| | std.dev | 2.6 | 1.6 | 1.0 | 4.1 | 0.3 | 8.7 |
| n u u | max | 5.0+ | 6.5- | 2.2+ | 13.1- | 1.7- | 26.6- |
| | avg | 1.0 | 1.3 | 0.3 | 3.4 | 0.1 | 7.3 |
| | std.dev | 1.2 | 1.6 | 0.5 | 3.8 | 0.2 | 8.0 |
| n u z | max | 6.9- | 35.6- | 2.0- | 33.6- | 1.3- | 38.7- |
| | avg | 0.5 | 5.7 | 0.1 | 3.8 | 0.1 | 4.2 |
| | std.dev | 0.7 | 10.2 | 0.2 | 7.8 | 0.1 | 7.3 |
| n z n | max | 11.6+ | 36.6- | 4.1+ | 34.5- | 1.3- | 38.8- |
| | avg | 0.6 | 5.9 | 0.2 | 4.0 | 0.1 | 4.5 |
| | std.dev | 0.9 | 10.6 | 0.4 | 8.0 | 0.2 | 7.5 |
| n z u | max | 7.1- | 35.5- | 2.5+ | 33.5- | 1.1- | 38.1- |
| | avg | 0.5 | 5.6 | 0.1 | 3.7 | 0.1 | 4.1 |
| | std.dev | 0.6 | 10.1 | 0.2 | 7.6 | 0.1 | 7.2 |
| n z z | max | 11.6- | 49.8- | 2.9- | 44.7- | 0.7- | 44.5- |
| | avg | 0.4 | **6.2** | 0.1 | 3.4 | 0.0 | 2.6 |

**Part 1 Table 6.7: Estimation Errors For Zipfian Boxes By The Uniform Scheme.**

| distr | stats | 5x5x5 | | 10x10x10 | | 20x20x20 | |
|---|---|---|---|---|---|---|---|
| | | ED | EW | ED | EW | ED | EW |
| u n n | max | 11.0+ | 7.4- | 4.6+ | 14.0- | 2.2- | 27.8- |
| | avg | 2.1 | 1.2 | 0.7 | 3.7 | 0.2 | 8.3 |
| | std.dev | 2.6 | 1.6 | 0.9 | 4.1 | 0.3 | 8.8 |
| u n u | max | 4.8+ | 6.6- | 2.1+ | 13.0- | 1.7- | 26.7- |
| | avg | 0.9 | 1.3 | 0.3 | 3.4 | 0.1 | 7.4 |
| | std.dev | 1.2 | 1.6 | 0.4 | 3.7 | 0.2 | 8.0 |
| u n z | max | 7.0- | 35.6- | 2.2+ | 33.5- | 1.2- | 38.8- |
| | avg | 0.5 | 5.7 | 0.1 | 3.8 | 0.1 | 4.2 |
| | std.dev | 0.7 | 10.1 | 0.2 | 7.7 | 0.1 | 7.3 |
| u u n | max | 4.7+ | 6.0- | 2.2+ | 12.5- | 1.0- | 26.4- |
| | avg | 0.9 | 1.3 | 0.3 | 3.4 | 0.1 | 7.4 |
| | std.dev | 1.2 | 1.5 | 0.4 | 3.8 | 0.2 | 8.1 |
| u u u | max | 0.5- | 5.1- | 0.3- | 11.7- | 0.6- | 25.8- |
| | avg | 0.1 | 1.4 | 0.0 | 3.1 | 0.0 | 6.6 |
| | std.dev | 0.1 | 1.5 | 0.0 | 3.5 | 0.1 | 7.5 |
| u u z | max | 6.5- | 34.2- | 1.6- | 32.5- | 0.5- | 37.4- |
| | avg | 0.6 | 5.4 | 0.1 | 3.5 | 0.0 | 3.8 |
| | std.dev | 0.8 | 9.7 | 0.2 | 7.4 | 0.1 | 7.0 |
| u z n | max | 7.4+ | 35.4- | 2.9+ | 33.4- | 1.0- | 38.0- |
| | avg | 0.5 | 5.6 | 0.1 | 3.7 | 0.1 | 4.1 |
| | std.dev | 0.7 | 10.0 | 0.2 | 7.6 | 0.1 | 7.2 |
| u z u | max | 6.4- | 34.1- | 1.5- | 32.5- | 0.7+ | 37.2- |
| | avg | 0.6 | 5.2 | 0.1 | 3.4 | 0.1 | 3.8 |
| | std.dev | 0.8 | 9.6 | 0.2 | 7.2 | 0.1 | 6.9 |
| u z z | max | 11.2- | 48.6- | 2.5- | 43.7- | 0.5- | 43.8- |
| | avg | 0.5 | 5.9 | 0.1 | 3.2 | 0.0 | 2.5 |
| | std.dev | 0.9 | 12.2 | 0.2 | 8.7 | 0.1 | 6.9 |

**Part 2 Table 6.7: Estimation Errors For Zipfian Boxes By The Uniform Scheme.**

| distr | stats | 5x5x5 | | 10x10x10 | | 20x20x20 | |
|---|---|---|---|---|---|---|---|
| | | ED | EW | ED | EW | ED | EW |
| z n n | max | 9.9+ | 36.5- | 3.7+ | 34.4- | 1.6- | 39.1- |
| | avg | 0.6 | 5.9 | 0.2 | 4.0 | 0.1 | 4.5 |
| | std.dev | 0.9 | 10.7 | 0.4 | 8.1 | 0.2 | 7.6 |
| z n u | max | 6.9- | 35.4- | 2.4- | 33.3- | 1.3- | 38.1- |
| | avg | 0.5 | 5.6 | 0.1 | 3.7 | 0.1 | 4.1 |
| | std.dev | 0.6 | 10.1 | 0.2 | 7.7 | 0.1 | 7.3 |
| z n z | max | 11.6- | 49.8- | 3.3- | 44.6- | 0.9- | 44.8- |
| | avg | 0.4 | 6.2 | 0.1 | 3.4 | 0.0 | 2.7 |
| | std.dev | 0.9 | 12.7 | 0.2 | 9.0 | 0.1 | 7.1 |
| z u n | max | 6.9- | 35.5- | 2.4- | 33.5- | 0.8- | 38.1- |
| | avg | 0.5 | 5.6 | 0.1 | 3.7 | 0.1 | 4.1 |
| | std.dev | 0.6 | 10.1 | 0.2 | 7.7 | 0.1 | 7.3 |
| z u u | max | 6.5- | 34.2- | 1.7- | 32.5- | 0.5- | 37.2- |
| | avg | 0.6 | 5.3 | 0.2 | 3.5 | 0.1 | 3.8 |
| | std.dev | 0.7 | 9.6 | 0.2 | 7.3 | 0.1 | 7.0 |
| z u z | max | 11.2- | 48.8- | 2.5- | 43.8- | 0.3- | 43.9- |
| | avg | 0.5 | 5.8 | 0.1 | 3.2 | 0.0 | 2.5 |
| | std.dev | 0.9 | 12.3 | 0.2 | 8.7 | 0.0 | 6.9 |
| z z n | max | 11.7- | 49.9- | 3.2- | 44.7- | 0.8- | 44.6- |
| | avg | 0.4 | 6.1 | 0.1 | 3.4 | 0.0 | 2.6 |
| | std.dev | 0.9 | 12.7 | 0.2 | 8.9 | 0.1 | 7.1 |
| z z u | max | 11.3- | 48.7- | 2.7- | 43.7- | 0.6- | 43.7- |
| | avg | 0.5 | 5.8 | 0.1 | 3.2 | 0.0 | 2.5 |
| | std.dev | 0.9 | 12.2 | 0.2 | 8.7 | 0.1 | 6.9 |
| z z z | max | 14.7- | **56.6-** | 2.9- | **50.1-** | 0.5- | **47.6-** |
| | avg | 0.3 | 5.4 | 0.1 | 2.8 | 0.0 | 1.8 |
| | std.dev | 1.0 | 13.0 | 0.2 | 9.0 | 0.1 | 6.7 |

**Part 3 Table 6.7: Estimation Errors For Zipfian Boxes By The Uniform Scheme.**

In order to find out how valid the results under the equi-depth columns in Table 6.7 are, we once again used the method of batch means. We generated 20 batches of 1000 query boxes each such that the volumes of the boxes in each batch had approximately a zipfian distribution. As before, the boxes were located uniformly over the tuple space. In each batch we estimated the number of tuples in each query box using the Uniform Scheme. For each of these batches, we calculated the average percentage deviation for each tuple distribution and each equi-depth histogram configuration. All statements that follow in this paragraph assume a confidence of 90%. Out of a total of 81 cases, the variance of the average was less than 5% of the average in 1 case, less than 7% of the average in 42 cases, less than 9% of the average in 72 cases. In seven other cases, the variance of the average was between 9% and 12% of the average. In between 16% and 17% of the average. However, the variances in these two cases were extremely small (0.008% and 0.016%). In fact, the variance never exceeded 0.19% in any of the 81 cases.

The objective of the last experiment was to demonstrate the effectiveness of the random sampling technique in building equal depth histograms. We chose a sample size of 8000, as this was the smallest multiple of 20 * 20 * 20. The results are presented in the three parts of Table 6.8. The format of Table 6.8 is identical to that of Table 6.4.

| distr | stats | 5x5x5 | 10x10x10 | 20x20x20 | Sample |
|---|---|---|---|---|---|
| | | ED | ED | ED | |
| n n n | max | **19.4+** | **5.8+** | 1.1+ | 1.1+ |
| | avg | **3.6** | **0.9** | 0.2 | 0.2 |
| | std.dev | 4.4 | 1.2 | 0.2 | 0.2 |
| n n u | max | 10.5+ | 3.6+ | 1.3- | 1.3- |
| | avg | 2.0 | 0.5 | 0.2 | 0.2 |
| | std.dev | 2.5 | 0.7 | 0.2 | 0.2 |
| n n z | max | 9.8+ | 2.7+ | 0.9+ | 0.9+ |
| | avg | 0.6 | 0.2 | 0.1 | 0.1 |
| | std.dev | 0.9 | 0.2 | 0.2 | 0.2 |
| n u n | max | 11.8+ | 5.0+ | 1.9+ | 0.9+ |
| | avg | 2.4 | 0.8 | 0.4 | 0.4 |
| | std.dev | 2.9 | 1.0 | 0.4 | 0.4 |
| n u u | max | 4.8+ | 2.7- | 1.2+ | 1.2+ |
| | avg | 1.0 | 0.4 | 0.2 | 0.2 |
| | std.dev | 1.3 | 0.4 | 0.2 | 0.2 |
| n u z | max | 5.8- | 1.7- | 1.0- | 1.0- |
| | avg | 0.5 | 0.3 | 0.2 | 0.2 |
| | std.dev | 0.6 | 0.3 | 0.2 | 0.2 |
| n z n | max | 11.2+ | 3.2+ | 0.9+ | 0.9+ |
| | avg | 0.6 | 0.3 | 0.2 | 0.2 |
| | std.dev | 0.9 | 0.4 | 0.2 | 0.2 |
| n z u | max | 7.4- | 2.8- | 1.1+ | 1.1+ |
| | avg | 0.4 | 0.2 | 0.2 | 0.2 |
| | std.dev | 0.6 | 0.2 | 0.2 | 0.2 |
| n z z | max | 10.8- | 1.5+ | 2.2+ | 2.2+ |
| | avg | 0.5 | 0.2 | 0.2 | 0.2 |
| | std.dev | 0.8 | 0.2 | 0.3 | 0.3 |

**Part 1 of Table 6.8: Estimation Errors By Sampling With The Uniform Scheme.**

Comparing the equi-depth columns of Table 6.7 with the corresponding ones in Table 6.8, we see that the sampling technique performs very well. We calculated the differences between the corresponding "avg" values in Table 6.7 and Table 6.8. The maximum difference between the corresponding "avg" entries were as follows: 0.513% (5x5x5), 0.509% (10x10x10), and 0.422% (20x20x20). Thus, the estimates obtained from the histograms built using the sample are well within the tolerance expected (1.82%[33]).

It is not a coincidence that the third column is identical to the fourth column in Table 6.8. Since the number of buckets (S = 8000) is equal to the sample size, each bucket consists of exactly one tuple. Let n denote the set of tuples (f-buckets) within any given query box. There will be no p-buckets for any query

| distr | stats | 5x5x5 | 10x10x10 | 20x20x20 | Sample |
|---|---|---|---|---|---|
| | | ED | ED | ED | |
| u n n | max | 10.7+ | 3.7+ | 1.1+ | 1.1+ |
| | avg | 2.0 | 0.5 | 0.2 | 0.2 |
| | std.dev | 2.5 | 0.6 | 0.2 | 0.2 |
| u n u | max | 5.3+ | 2.7+ | 1.4+ | 1.4+ |
| | avg | 1.2 | 0.6 | 0.3 | 0.3 |
| | std.dev | 1.4 | 0.6 | 0.3 | 0.3 |
| u n z | max | 6.6- | 2.4+ | 1.3+ | 1.3+ |
| | avg | 0.4 | 0.2 | 0.2 | 0.2 |
| | std.dev | 0.6 | 0.2 | 0.2 | 0.2 |
| u u n | max | 5.0+ | 2.2+ | 1.2- | 1.2- |
| | avg | 1.1 | 0.4 | 0.2 | 0.2 |
| | std.dev | 1.3 | 0.4 | 0.2 | 0.2 |
| u u u | max | 1.9- | 2.6- | 2.0- | 2.0- |
| | avg | 0.6 | 0.6 | 0.5 | 0.5 |
| | std.dev | 0.6 | 0.6 | 0.5 | 0.5 |
| u u z | max | 6.3- | 1.5- | 1.0+ | 1.0+ |
| | avg | 0.6 | 0.2 | 0.1 | 0.1 |
| | std.dev | 0.7 | 0.2 | 0.1 | 0.1 |
| u z n | max | 7.1- | 2.5- | 1.0+ | 1.0+ |
| | avg | 0.5 | 0.2 | 0.2 | 0.2 |
| | std.dev | 0.7 | 0.2 | 0.2 | 0.2 |
| u z u | max | 6.7- | 2.4- | 1.1+ | 1.1+ |
| | avg | 0.5 | 0.2 | 0.2 | 0.2 |
| | std.dev | 0.7 | 0.2 | 0.2 | 0.2 |
| u z z | max | 9.8- | 1.6+ | 2.2+ | 2.2+ |
| | avg | 0.5 | 0.2 | 0.2 | 0.2 |
| | std.dev | 0.8 | 0.2 | 0.3 | 0.3 |

**Part 2 of Table 6.8: Estimation Errors By Sampling With The Uniform Scheme.**

[33] $0.0182 \approx (1063 * (0.05)^2/8000)^{0.5}$.

| distr | stats | 5x5x5 | 10x10x10 | 20x20x20 | Sample |
|---|---|---|---|---|---|
| | | ED | ED | ED | |
| z n n | max | 9.7+ | 2.6- | 0.9+ | 0.9+ |
| | avg | 0.6 | 0.3 | 0.2 | 0.2 |
| | std.dev | 0.9 | 0.3 | 0.2 | 0.2 |
| z n u | max | 7.1- | 2.3- | 1.0- | 1.0- |
| | avg | 0.5 | 0.2 | 0.1 | 0.1 |
| | std.dev | 0.6 | 0.2 | 0.2 | 0.2 |
| z n z | max | 10.7- | 2.1- | 0.9- | 0.9- |
| | avg | 0.4 | 0.1 | 0.1 | 0.1 |
| | std.dev | 0.8 | 0.2 | 0.1 | 0.1 |
| z u n | max | 7.0- | 2.6- | 1.2+ | 1.2+ |
| | avg | 0.4 | 0.3 | 0.2 | 0.2 |
| | std.dev | 0.6 | 0.3 | 0.3 | 0.3 |
| z u u | max | 6.6- | 2.0- | 1.2+ | 1.2+ |
| | avg | 0.5 | 0.2 | 0.2 | 0.2 |
| | std.dev | 0.7 | 0.2 | 0.3 | 0.3 |
| z u z | max | 10.5- | 1.7- | 0.9- | 0.9- |
| | avg | 0.5 | 0.2 | 0.1 | 0.1 |
| | std.dev | 0.9 | 0.2 | 0.1 | 0.1 |
| z z n | max | 11.7- | 2.5- | 0.8+ | 0.8+ |
| | avg | 0.4 | 0.2 | 0.2 | 0.2 |
| | std.dev | 0.8 | 0.2 | 0.2 | 0.2 |
| z z u | max | 11.4- | 2.3- | 1.2+ | 1.2+ |
| | avg | 0.5 | 0.1 | 0.1 | 0.1 |
| | std.dev | 0.9 | 0.2 | 0.2 | 0.2 |
| z z z | max | 13.2- | 1.0+ | 1.5+ | 1.5+ |
| | avg | 0.3 | 0.1 | 0.1 | 0.1 |
| | std.dev | 0.9 | 0.1 | 0.1 | 0.1 |

**Part 3 of Table 6.8: Estimation Errors By Sampling With The Uniform Scheme.**

box since each bucket is consists of a single tuple. Each bucket, by definition, has exactly $\frac{N}{S}$ tuples in it. Therefore, the number of tuples estimated to be within a query box, by the Uniform Scheme is given by n $\frac{N}{S}$. On the other hand, the fraction of tuples in the sample that lie within a query box is $\frac{n}{S}$. Since we assumed that the fraction of tuples in the sample that lie within the query box is the same as the fraction of tuples in the population that also lie within the query box, the estimated number of tuples in the population that lie within the query box is equal to $\frac{n}{S}$ N.

# CHAPTER 7

## SUMMARY

In this dissertation we have demonstrated systematic techniques for the optimization of multiple-disjunct queries in a relational database system. From our review of the literature in query optimization, it was clear that there has been little work done on optimizing queries involving multiple disjuncts. This, in spite of the vast amount of research activity in the area of query optimization for over a decade, is somewhat surprising. Since several aspects of the optimization of single-disjunct queries were well known, the standard approach to optimizing a multiple-disjunct query was to optimize each disjunct separately. The result of the query was then obtained by taking the union of the results from each disjunct. However, the most important 'lesson' that we have learned from this thesis is that optimizing each disjunct independently can be very inefficient. Substantial savings in cost may be realized when the disjuncts are optimized together rather than when they are optimized separately.

The problem of optimizing multiple-disjunct queries can be broken up into two different subproblems. The first subproblem deals with the optimization of multiple-relation, multiple-disjunct queries. Optimizing a multiple-relation multiple-disjunct query consists of merging disjuncts, whenever possible, into the minimum number of terms. Assuming no relevant indices on the individual relations, the number of scans on each relation in the query is equal to the number of result terms. The problem of minimizing the number of terms was formulated as the problem of covering a merge graph with the minimum number of complete merge graphs. The latter problem is, in general, NP-complete. We developed polynomial time algorithms for a special class of merge graphs called simple merge graphs and presented a heuristic for merge graphs that are not simple.

The second subproblem dealt with the optimization of single-relation, multiple-disjunct queries. Merging disjuncts in a multiple-relation query into terms has the effect of bringing together the disjuncts of individual relations. These disjuncts are then optimized together using the appropriate access paths. When optimizing single-relation disjuncts together, the objective was to cover the solution space with a set of

subspaces such that the sum of the costs of the subspaces was minimized. Covering the solution space with subspaces of minimum cost can be formulated as finding the cheapest vertex cover in a hypergraph.

The problem of finding the cheapest vertex cover in a m-dimensional ($m \geq 3$) hypergraph is NP-complete. The Bar-Yehuda approximation algorithm guarantees a vertex cover of cost no greater than $m * W_{opt}$, where $W_{opt}$ is the cost of the optimal vertex cover. We developed a new approximation algorithm that performs much better, on the average, than the Bar-Yehuda algorithm on random graphs and random hypergraphs over a wide range of densities.

Finally, we discussed the efficacy of equi-depth, multi-dimensional histograms in predicting the number of tuples within a multi-dimensional query box. Equi-depth histograms 'capture' the notion of distribution very elegantly. We can decrease the error in estimating selectivity factors to an arbitrary extent by increasing the number of equi-depth buckets. We presented a very efficient algorithm to generate equi-depth histograms and used a variant of the R-tree to store them in a main memory data structure. Using the Uniform Scheme of estimation, we showed experimentally that we can estimate selectivity factors with a high degree of accuracy. We also demonstrated the usefulness of the Random Sampling Technique in building equi-depth histograms at a very low cost.

### 7.1. Future Research Directions

Having come up with techniques for optimizing multiple-disjunct queries, the next logical step is to design a benchmark, similar to the Wisconsin Benchmark [Bitton83], for multiple-disjunct queries. It will certainly be interesting to test existing query optimizers against the benchmark. Designing a benchmark for multiple-disjunct queries will also help in evaluating how well the new approximation algorithm performs on typical multi-disjunct queries.

In this thesis, we have not concerned ourselves with strategies for optimizing joins. One of the key problems that still remains unsolved to a great extent is the determination of the optimal join order within a single disjunct, especially when there is more than one join operation in the query. The total number of join orders is exponential in the number of relations to be joined even when the query graph is linear. However, using the techniques of dynamic programming, we can prune the number of join orders such that we need consider only $O(n^2)$ orders. We then intend to use sampling techniques to select the optimal join

order from among these $O(n^2)$ join orders. The idea is to keep a small sample ($\approx$ 1000 tuples) of each relation in main memory. Computing the join for each of the $O(n^2)$ join orders using the samples should give us an estimate of the cost for each join order. The main point here is that, with sampling, we can avoid assuming join selectivities and distribution of join column values.

Using equi-depth, multi-dimensional histograms we were able to estimate the number of tuples in a query box with a high degree of accuracy. Besides being able to compute selectivity factors accurately, a query optimizer also needs an estimate of the number of pages that will be fetched from secondary storage in order to retrieve all the tuples within a query box. Estimating the number of pages accessed when using a clustered index on a single-attribute is straightforward. The System-R query optimizer [Selinger79] assumes that the number of pages accessed when using a non-clustered index is equal to the number of tuples retrieved. We could adopt the same solution when estimating the number of pages accessed when using a multi-attribute index such as the KDB-tree or the Grid file index. However, this assumption is very conservative. We would like to design a data structure, for estimating the number of pages that will be fetched for a given query box, that can be used in conjunction with an index structure such as the K-D-B Tree or the Grid file.

# CHAPTER 8

## REFERENCES


Aho79.
A.V. Aho, Y. Sagiv, and J.D. Ullman, "Equivalence of Relational Expressions," *SIAM Journal of Computing* **8**(2) pp. 218-246 (1979).

Bar-Yehuda81.
R. Bar-Yehuda and S. Even. "A Linear-Time Approximation Algorithm for the Weighted Vertex Cover Problem," *Journal of Algorithms* **2** pp. 198-203 (January 1981).

Bayer72.
R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica* **1** pp. 173-189 (1972).

Bazaraa77.
M.S. Bazaraa and J.J. Jarvis, *Linear Programming and Network Flows*, John Wiley and sons, New York (1977).

Bernstein81.
P.A. Bernstein, N. Goodman, E. Wong, C. Reeve, and J.B. Rothnie, "Query Processing in a System for Distributed Databases (SDD-1)," *ACM Trans. on Database Systems* **6**(4) (Dec. 1981).

Bernstein81a.
P.A. Bernstein and D.M.W. Chiu, "Using semi-joins to solve relational queries," *ACM Journal* **28** pp. 25-40 (January 1981).

Bernstein81b.
P.A. Bernstein and N. Goodman, "The power of natural semijoins," *SIAM journal of Computing* **10**(4) pp. 751-771 (1981).

Bitton83.
D. Bitton, D.J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach," Computer Sciences Technical Report, (526) University of Wisconsin, (Dec. 1983).

Blasgen77.
M. Blasgen and K. Easwaran, "Storage and Access in Relational Databases," *IBM Systems Journal* **16**(4) pp. 751-771 (1981).

Chandra77.
A.K. Chandra and P.M. Merlin, "Optimal Implementation of Conjunctive Queries in Relational Data Bases," *Proc. of the 9th Annual ACM symposium on Theory of Computation*, pp. 77-90 (May 1977).

Codd70.
E.F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Comm. of the ACM* **13**(6) pp. 377-387 (June 1970).

Codd72.
E.F. Codd, "Relational Completeness of Database Sublanguages," pp. 65-98 in *Data Base Systems*, ed. R. Rustin, Prentice-Hall, New York (1972).

Comer79.
D. Comer, "The Ubiquitous B-Tree," *Computing Surveys* 11(2)(June 1979).

Dayal87.
U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers", *Proc. Conf. Very Large Data Bases*, (September 1987).

DeWitt84.
D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proc. ACM SIGMOD Conf.*, pp. 1-8 (June 1984).

DeWitt85.
D.J. DeWitt and R. Gerber, "Multiprocessor Hash-Based Join Algorithms," *Proc. Conf. Very Large Data Bases*, (August 1985).

Fagin79.
R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, "Extendible Hashing : A fast access method for dynamic files," *ACM Trans. on Database Systems* Vol. 4(3) pp. 315-344 (Sept. 1979).

Garey76.
M.R. Garey, D.S. Johnson, and L. Stockmeyer, "Some simplified NP-complete graph problems," *Theoretical Computer Science* 1 pp. 237-267 (1976).

Garey79
M.R. Garey and D.S. Johnson, *Computers and Intractability*, Freeman & Co., New York (1979).

Gibbons76.
J.D. Gibbons, *Nonparametric methods for quantitative analysis*, Holt, Rinehart and Winston, New York (1976).

Goodman81.
J.R. Goodman, "An Investigation of Multiprocessor Structures and Algorithms for Database Management," *Technical Report UCB/ERL, M81/33*, (May 1981).

Gotlieb75.
L.R. Gotlieb, "Computing joins of relations," *Proc. of ACM SIGMOD*, pp. 55-63 (May 1975).

Gries71.
D. Gries, *Complier construction for digital computers*, Wiley, New York (1971).

Guttman84.
A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Conf.*, pp. 47-57 (June 1984).

Hammer80.
M. Hammer and S.B. Zdonik, "Knowledge-based query processing," *Proc. of the 6th International conf. on Very Large Databases*, pp. 137-147 (October 1980).

Ibaraki84.
T. Ibaraki and T. Kameda, "On the optimal nesting order for computing N-Relational Joins," *Transactions on Database Systems* 9(3) pp. 482-502 (September 1984).

Jarke84.
M. Jarke and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys* 16(2) pp. 111-152 (June 1984).

Kerschberg82.

L. Kerschberg, P.D. Ting, and S.B. Yao, "Query optimization in a star computer network," *ACM Trans. on Database Systems* 7(4) pp. 678-711 (Dec. 1982).

Kim82.
W. Kim, "On optimizing an SQL-like nested query," *ACM Trans. on Database Systems* 7(3) pp. 443-469 (Sept. 1982).

King81.
J.J. King, "QUIST: A system for semantic query optimization in relational databases," *Proc. of the 7th International conf. on Very Large Databases*, pp. 510-517 (Sept. 1981).

Kitsuregawa83.
M. Kitsuregawa, H. Tanaka, and T. Moto-oka, "Application of hash to data base machine and its architecture," *New generation computing* 1(1)(1983).

Klug82.
A. Klug "Equivalence of relational algebra and relational calculus query languages having aggregate functions," *ACM Journal* 29(3) pp. 699-717 (July 1982).

Krishnamurthy86.
R. Krishnamurthy, H. Boral, and C. Zaniolo, "Efficient Processing of Nonrecursive Queries," *Technical Report, (DB-022-86)MCC*, (Feb. 1986).

Litwin80.
W. Litwin, "Linear Hashing : A New Tool For File and Table Addressing," *Proc. Conf. Very Large Data Bases*, (October 1980).

Lovasz79.
L. Lovasz, *Combinatorial Problems and Exercises*, North Holland, Amsterdam (1979).

Nievergelt84.
J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: An adaptable, symmetric multikey file structure," *ACM Trans. on Database Systems* 9(1) pp. 38-71 (March 1984).

Pruhs87.
Kirk Pruhs, "Finding a Minimum Cover of Complete Merge Graphs in a Merge Graph is NP-complete," *Private Communication*, University of Wisconsin, (Feb. 1987).

Robinson81.
J. Robinson, "The K-D-B Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. of the ACM SIGMOD*, (1981).

Sargent76.
R.G. Sargent, "Statistical Analysis of Simulation Output Data," *Proc. ACM Symp. on the Simulation of Computer Systems IV*, pp. 39-50 (August 1976).

Schkolnik85.
M. Schkolnik and P. Tiberio, "Estimating the Cost of Updates in a Relational Database," *ACM Trans. on Database Systems* 10(2) pp. 163-179 (June 1985).

Sedgewick84.
R. Sedgewick, *Algorithms*, Addison Wesley, Reading, Massachusetts (1984).

Selinger79.
P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proc. ACM SIGMOD Conf.*, (June 1979).

Shapiro84.
G.P. Shapiro and C. Connell, "Accurate estimation of the number of tuples satisfying a condition," *Proc. of ACM SIGMOD*, pp. 256-276 (June 1984).

Smith75.
J.M. Smith and P.Y.T. Chang, "Optimizing the performance of a relational database interface," *Comm. of the ACM* **18**(10) pp. 568-579 (Oct. 1975).

Stonebraker76.
M. Stonebraker, E. Wong, P. Kreps, and G.D. Held, "The Design and Implementation of INGRES," *ACM Trans. on Database Systems* **1**(3) pp. 189-222 (Sept. 1976).

Ullman82.
J.D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, MD. (1982).

Valduriez84.
P. Valduriez and G. Gardarin, "Join and Semijoin Algorithms for a Multiprocessor Database Machine," *Trans. on Database Systems* **9**(1) p. 133 (March 1984).

Valduriez85.
P. Valduriez, "Join Indices," *MCC Technical Report DB-052-85*, (1985).

Wong76.
E. Wong and K. Youssefi, "Decomposition - A Strategy for Query Processing," *ACM Trans. on Database Systems* **1**(3) pp. 223-241 (Sept. 1976).

Yao79.
S.B. Yao, "Optimization of query evaluation algorithms," *ACM Trans. on Database Systems* **4**(2) pp. 133-155 (June 1979).

Youssefi79.
K. Youssefi and E. Wong, "Query processing in a relational database system," *Proc. of the 5th International Conf. on Very Large Data Bases*, pp. 409-417 (Oct. 1979).

Zipf49.
G. K. Zipf, *Human Behavior and the Principle of Least Effort*, Addison-Wesley, Cambridge, M.A. (1949).

Zloof77.
M.M. Zloof, "Query-by-example: A data base language," *IBM Syst. J.* **16**(4) pp. 324-343 (1977).