Processing Queries By Linear Constraints *

Jonathan Goldstein
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706
jgoldst@cs.wisc.edu

Raghu Ramakrishnan
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706
raghu@cs.wisc.edu

Uri Shaft
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706
uri@cs.wisc.edu

Jie-Bing Yu
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706
jiebing@cs.wisc.edu

Abstract

The emergence of several new application domains for databases has introduced the need for more efficient complex query handling than databases currently support. These application domains include *On-Line Analytical Processing* (OLAP), *Geographical Information Systems* (GIS), and scientific databases.

This paper focuses attention on a form of selection query, expressible in SQL but not evaluated efficiently by current DBMSs, with wide applicability in these new problem domains. We introduce a processing strategy for this class of queries, which we call *queries by linear constraints* (QBLC). This processing strategy can be implemented with a wide variety of multidimensional indexing structures that include the R-Tree variants, the k-d-B-Tree, the Buddy-Tree, and many more. Note that any processing strategy meant for general database use must **guarantee** that all correct answers are returned. Therefore, all numerical techniques we employ uphold this guarantee. Thus the most distinguishing characteristic of this processing strategy is its safe handling of numerical error (which can result in the dismissal of valid answers).

This paper presents several theoretical results about our processing strategy, and the results of several experiments which show that the processing cost of selection queries by linear constraints can be reduced dramatically by using our processing strategy.

* This work was partially supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award, NASA Research Grant NAGW-3921, and ORD contract 144-ET33.

Copyright ©1997 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM Inc., fax +1 (212) 869-0481, or (permissions@acm.org).

1 Introduction

1.1 Description of Queries By Linear Constraints (QBLC)

The emergence of several new application domains for databases has highlighted the need for more efficient complex query handling than databases currently support. The application domains include *On-Line Analytical Processing* (OLAP), *Geographical Information Systems* (GIS), and scientific databases.

This paper discusses a form of selection query, expressible in SQL but not handled efficiently by current DBMSs, with wide applicability in these new problem domains. The following is an example of such a query: "Retrieve the names of all companies from relation *Companies(Name, NumEmployees, ..., PricePerShare, EarningsPerShare)* that have a price-earnings ratio less than or equal to 10, and have at least 50 employees." The query, when expressed in SQL, is the following:

```
SELECT Name  FROM Companies
WHERE  (PricePerShare - 10*EarningsPerShare <= 0)
       AND (NumEmployees >= 50)
```

This is a selection query whose selection criterion is a conjunction of two constraints. We call these constraints *linear constraints* because they are built as a comparison of a *linear combination of attributes* to a constant.

Many database applications deal with spatial data. These commonly use queries of the form: "Retrieve all objects that overlap a given convex region". In most cases the convex region is (or can be) specified by using a conjunction of linear constraints. A GIS example would be: "Retrieve all restaurants along a specific road segment" (e.g., along I90 from Madison to Chicago).

Suppose the road segment has two end-points $(X1, Y1)$, $(X2, Y2)$ (with $X1 < X2$ and $Y1 < Y2$). Suppose the *Restaurants* relation has attributes X, Y for specifying its location. A possible SQL query would be:

```
SELECT *  FROM Restaurants
WHERE  (X >= X1 - e) AND (X <= X2 + e)
   AND [X*(Y1-Y2) + Y*(X2-X1) <= X2*Y1 - X1*Y2 + e]
   AND [X*(Y1-Y2) + Y*(X2-X1) >= X2*Y1 - X1*Y2 - e]
```

We use a small constant e in the query to get points X, Y that are close to the road. Figure 1 shows the half-plane

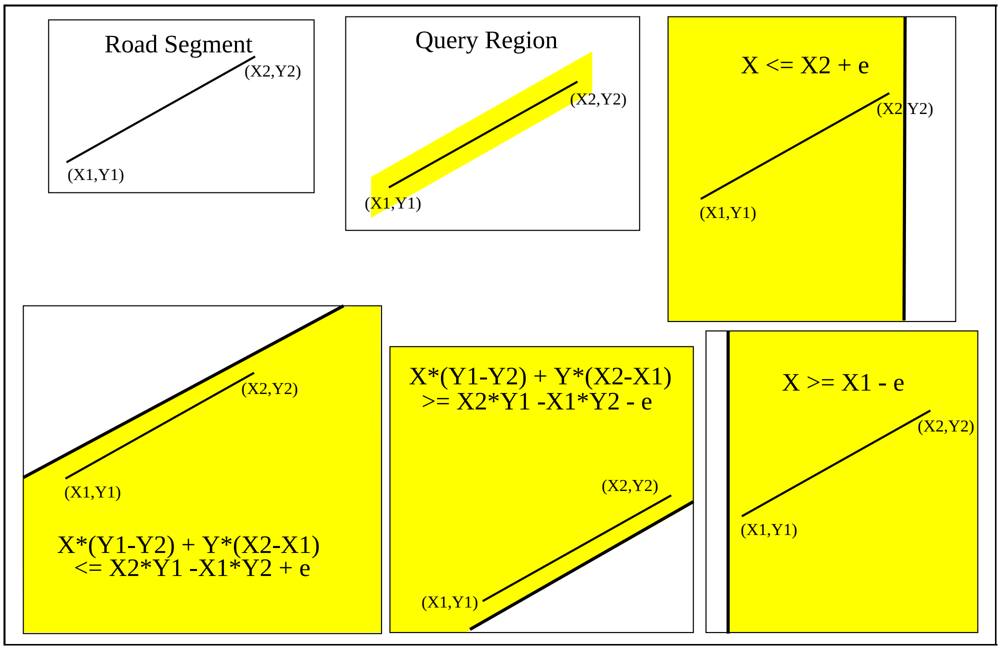

Figure 1: Query shape around a line segment.

created by each constraint and the shape of the query region. We call such queries *Queries By Linear Constraints* (QBLC). This paper argues that using and optimizing QBLC will significantly enhance current database systems. In addition, any processing strategy meant for general database use must **guarantee** that all correct answers are contained in the answer. Therefore, all numerical techniques we employ uphold this guarantee.

1.2 A Processing Strategy for QBLC

Efficient processing of QBLC, like other kinds of selection queries, involves using an indexing structure. The data space of the indexing structure is multidimensional since we want to employ all the attributes specified in the selection criterion. All indexing structures involve cutting the data space into smaller regions, where the choice of region divisions is usually based on data distribution. The shape of these subdivided regions differs among different indexing structures. A QBLC query asks for all data objects that overlap (or fall inside) a region of the data space. Algorithmically, this requires a test that determines whether a particular section of data space overlaps the query region.

Our solution is designed to work with indexing structures that use *hyper-rectangles* as regions. Such structures are variants of the R-Tree [7, 1, 15], the k-d-B-Tree [14] (described further in section 4), the Buddy-Tree [8, 9], and many more. We chose to implement our strategy with the R^*-Tree [1] (which is a variant of the R-Tree). These indexing structures support *range selection queries*. These queries are conjunctions of *range constraints*. Range constraints are linear constraints that involve only one attribute. For example, the first and second constraints of the *Restaurants* example from Section 1.1 are range constraints. The resulting query region, when all constraints are range constraints, is a hyper-rectangle. The search algorithms used by these indexing structures need to repeatedly test if two hyper-rectangles overlap (as a test for whether the query region overlaps an indexing region). Note that these tests contain only equality and inequality tests for the hyper-rectangle bounds, and are, therefore, immune to effects of numerical error.

We replace the overlap test between two hyper-rectangles in the indexing structures' search algorithms with one that determines whether a conjunction of linear constraints overlaps a hyper-rectangle.

For years linear programming researchers have been studying a related problem (e.g. see [2, 4]). Indeed, the overlap test can be framed as a linear program! While this seems like a reasonable solution at first, there is a serious problem. Linear programming solutions involve a complex series of floating point operations. Since each of these operations can result in error, the accumulated error can produce invalid answers that result in rejecting valid data. Clearly a query processing technique that does not generate all answers is unacceptable. Modifying algorithms such as "Simplex" or "Karmarkar's" to avoid dropping valid data will result in complex and costly algorithms, even for small numbers of constraints. We have therefore developed more efficient tests that are numerically safe (guaranteed not to drop valid answers) for the problem of testing if a QBLC region overlaps an MBH.

Computational geometry [13, 11] seemingly provides us with another possible source of algorithms for our overlap test. If one restricts the queries to finite hyper-volume, the conjunction of linear constraints with the bounding box from the index structure describes a convex polyhedron. Determining if the set of vertices that describe this polyhedron is empty is equivalent to solving the intersection problem. Such an algorithm is presented in [11]. Unfortunately it scales factorially with respect to dimensionality. It is easy to see why it is not possible to solve the problem this way in better than exponential time when one considers that the result, in many cases, is a hyper-rectangle. Since the number of vertices in a hyper-rectangle scales exponentially with the number of dimensions, the output scales exponentially with the number of dimensions.

If we assume we had a vertex representation of the query region, a polyhedron-polyhedron intersection test could be used. Unfortunately, these tests suffer from many problems. First, the vertex representation of the bounding box from the index structure is less efficient than constraint representations. The number of vertices in the hyper-rectangle from the index structure scales exponentially with respect to dimensionality while the number of faces (constraints) scales linearly. In addition, some algorithms, like Günther and Wong's [6], scale exponentially with the number of dimensions.

1.3 Summary of Main Results

We present several theoretical results about the correctness and efficiency of our algorithms. We have also implemented our processing strategy using R*-Trees and performed many experiments on various data sets. Some data sets were artificial while others were taken from real applications. We tested the effect of many parameters on the performance of the processing strategies. These include: the query's *shape*, *size*, *complexity*, and the type of data used in the database. As a general trend we observed that the I/O cost benefit of our algorithms for a query is dependent on the ratio between the query's area (volume) and the area (volume) of its bounding-box. For example, if that ratio is 0.1 it means that our processing strategy uses about an order of magnitude less I/O than other strategies currently implemented in DBMSs. A very important aspect of the experiments is that when the number of constraints in the query is less than 100, which is far more than needed for most if not all queries, the overhead of our approach is less than 1% (even with a very slow CPU). Thus, while the performance gains can be considerable, the potential for doing worse, in comparison with the standard approach is negligible.

As an additional experiment we implemented the strategy in the Paradise [1] GIS. It was very easy to install the changes to R*-Trees in Paradise and the results on testing the strategy in Paradise demonstrated that the gains we observed in our experimental environment were similar to the gains observed in the real system. For details see [5].

2 Description of Our Solution

2.1 Issues in Modifying Indices' Search Algorithms

In addition to describing the R-Tree, this section discusses the modifications to R-Tree query execution needed to support QBLC. In this section, for convenience, we discuss R-Trees only. We briefly discuss some other indexing structures in Section 4.

2.1.1 The R-Tree indexing structure

R-Trees [7, 1, 15] are a family of height-balanced trees designed specifically for indexing multi-dimensional spatial data. An R-Tree hierarchically decomposes the data space into progressively smaller hyper-rectangular regions. Each node (a disk page) in the R-Tree is either an internal node or a leaf node. Internal nodes' entries point to children, which represent sub-regions of the original internal node. Internal nodes' entries also contain the *Minimum Bounding Hyper-rectangle* (MBH) of space represented by the child. Entries in leaf nodes contain the object IDs of the actual spatial objects (or, optionally, the objects themselves), and the MBH of the objects. Note that two entries in the same internal node may contain MBHs that overlap. All MBHs are oriented orthogonally with respect to the (fixed) axes for the R-Tree. The above description concerns R-Tree structure and is applicable to any of the variants of the R-Tree, including R^+-Trees [15] and R*-Trees [1]. The variants were created to reduce search time through improved space utilization, and reduced overlap. We will not discuss these variations further, since they do not affect our modification of the search algorithm.

2.1.2 Performing QBLC over the R-Tree

The R-Tree query algorithm is designed to retrieve all objects in the R-Tree that spatially overlap a given hyper-rectangle H. This is accomplished by a depth-first search of all nodes that spatially overlap H.

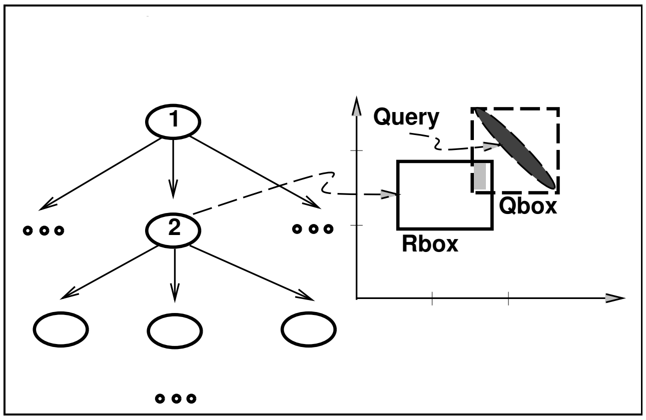

Figure 2: Search path in an R-Tree

At the leaf level, MBHs of objects are examined to determine which objects spatially overlap. More formally, the standard algorithm can be stated as follows:

```
DoQuery(RTreeNode:Node, HyperRectangle:Q)
for each entry E of Node
  if E.MBH overlaps Q
    if Node is a leaf
      Output E
    else
      DoQuery(E.Node, Q)
```

Figure 2 depicts the application of the overlap test within an R-Tree; the bounding box for the query is intersected with the bounding box for the entry in the tree. Modifying this to support QBLC is simple. The type of Q is changed to a conjunction of linear constraints, and the test used to evaluate if $E.MBH$ overlaps Q is replaced with one of the tests discussed in Sections 2.3 and 2.4. This ensures that the set of visited nodes is exactly the set of nodes that represent regions which spatially overlap the QBLC query region.

2.2 Formal Problem Definition for the Overlap Test

In the rest of this Section we develop efficient algorithms for testing whether a bounding box ($E.MBH$) overlaps a conjunction of linear constraints (Q). This section discusses notations and some desired properties of the test while Sections 2.3 and 2.4 present two algorithms that have these properties.

[1] Paradise [3] is a prototype object-relational DBMS for GIS applications developed at University of Wisconsin, and is built on top of the SHORE storage manager.

In order to discuss the test more clearly, it is necessary to give a more precise description of the problem. The test has two inputs. The first input, referred to as B throughout the paper, is an n dimensional hyper-rectangle orthogonal to the axes. B corresponds to a subdivision in the indexing structure and consists of a high and low bound on each of the n dimensions of the data. The following notation will be used in discussing the bounds on B throughout the paper:

$$B = [l_1, h_1] \times [l_2, h_2] \times \cdots \times [l_n, h_n] \quad \text{such that}$$

$$l_1, .., l_n, h_1, .., h_n \in \Re$$

The second input is a set of m linear constraints in n dimensions. These linear constraints are the conjunction of the linear constraints from the query itself. This set of linear constraints will be referred to as L while the geometric shape corresponding to L will be referred to as P. The linear constraints are given by $\vec{A}_1, .., \vec{A}_m \in \Re^n$ and $C = (c_1, .., c_m) \in \Re^m$. Constraint P_i is defined as

$$P_i = \left\{ \vec{X} \in \Re^n \,\middle|\, \vec{A}_i \cdot \vec{X} \geq c_i \right\}$$

The corresponding query region is $P = \bigcap_{i=1}^{m} P_i$. Sometimes, a bounding hyper-rectangle of P is given with the query. We denote it by B_P. If it is not given with the query we can always use the bounding hyper-rectangle of the entire indexing structure (MBH of the root of the tree) as a safe B_P.

The output of the algorithm is either "Yes" or "No". A "No" means that the query region and the hyper-rectangle definitely don't overlap (i.e. $B \cap P = \emptyset$). A "Yes" means that the query region and the hyper-rectangle may overlap (i.e. it is possible that $B \cap P \neq \emptyset$). Note that a "Yes" does not mean that the query region and hyper-rectangle definitely overlap. A "Yes" answer given for a query region and hyper-rectangle which don't overlap is called a false positive. An algorithm that returns fewer false positives than another is called *more accurate*.

The motivation for allowing false positives comes from two observations. First, false positives only result in extra answers to the query. These extra answers can be eliminated using post-processing. In addition, we show that allowing only a few false positives results in a much faster overlap test. False negatives, or answering "No" when the query region and hyper-rectangle overlap, are disallowed because they result in excluding valid answers to the query, an unrecoverable loss. We call an algorithm **safe** if it does not return false negatives. Obviously, only safe algorithms can be used for query processing in a general purpose DBMS..

2.3 The Simple Test

The simple test is based on the observation that if $\exists i$ s.t. $B \cap P_i = \emptyset$ then $B \cap P = \emptyset$. The essence of the simple test is, therefore, to find a constraint that cannot be satisfied by any point in B; if so, we can be assured that the correct answer to return is "No". The test returns "Yes" at all other times. Another way to phrase the simple test is the following: "Return "Yes" if and only if each constraint can be satisfied by some point inside the hyper-rectangle B."

The algorithm described so far may be made more accurate with a straightforward refinement: it can be performed after clipping B with B_P. The simple test with this refinement is shown in Figure 3.

Testing if $B' \cap P_i = \emptyset$ needs to be explained further since it is a non-trivial operation. In this explanation, we denote

1. $B' := B \cap Bp$
2. **for** $i := 1$ **to** m **do**
   a. **if** $B' \cap P_i = \emptyset$ **then return** "No"
3. **return** "Yes"

Figure 3: The simple test.

$B' = [l'_1, h'_1] \times \cdots \times [l'_n, h'_n]$. To test if $B' \cap P_i = \emptyset$, we find the point $\vec{z}$, inside B', that maximizes $\vec{A}_i \cdot \vec{X}$. If $\vec{z}$ is outside the half-space P_i, then every other point in B' is outside P_i so $B' \cap P_i = \emptyset$. More formally, Step 2(a) of the simple test can be done as follows:

1. Define $\vec{z} = (z_1, .., z_n)$ as follows:
   if $A_{ij} > 0$ **then** $z_j = h'_j$ **else** $z_j = l'_j$
2. $d := c_i - \vec{A}_i \cdot \vec{z}$
3. **if** $d > 0$ **then return** "No".

Let us consider whether the algorithm is safe. The only place where numerical error can occur is the dot product in Step 2. Step 2 expands to

$$d := c_i - A_{i1} \cdot z_1 + A_{i2} \cdot z_2 + \cdots + A_{in} \cdot z_n$$

If we naively implemented the dot product, we would simply do a series of multiplications and then add all the results to the accumulator d. Assuming that the mantissa of the multiplicands are of equal length m, and the exponents of size at most e, the result of the multiplication is a float with mantissa length at most $2m$ and exponent size of at most $e+1$. These results can be represented by more accurate floating point numbers. Therefore the real difficulty is in performing the additions. In Section 2.5, we describe a method of adding n IEEE floating point numbers in a way that bounds the error on the result. This error bound can then be corrected conservatively. This addition algorithm uses $\mathcal{O}(n)$ floating point operations and $\mathcal{O}(n \log n)$ integer operations (see Section 2.5 and Theorem 2.3). In the full version of this paper, we will present an addition algorithm with only $\mathcal{O}(n)$ operations.

Using the algorithm presented in Section 2.5 and Theorem 2.3 we have the following result about the simple test, stated in terms of parameters (e.g., n, m) and variables (e.g., B) defined in Section 2.2:

Theorem 2.1 :

1. *The simple test is $\mathcal{O}(mn)$ (which is linear in the input), and never yields a false negative.*
2. *For two dimensions ($n = 2$), if B_P is a minimal box that contains P, the simple test never yields a false positive.*
3. *If B is a point (i.e., $\forall i \quad l_i = h_i$), the simple test never yields a false positive.*

Part 1 of Theorem 2.1 proves that the simple test is an efficient overlap test. Part 2 of Theorem 2.1 proves the suitability of the simple test for 2D applications like GIS and chip design, where tight bounds for B_P are usually available. In these application domains, the simple test is ideal. Part 3 of Theorem 2.1 shows that the simple test will be able to exactly determine if a point overlaps a linear constraint query, even if no B_P is available. This is especially relevant for high dimensional data sets, which are usually point data.

In summary, the simple test is extremely suitable for GIS and some CAD databases. In addition, this test is ideal for testing point data (e.g., the leaf level of an R-Tree over point data). While this algorithm may be suitable for other problem domains, we don't know any theoretical way to quantify the likelihood of false positives. As a result, we examine the likelihood of false positives through experimentation in 3.2.

2.4 The Clipping Test

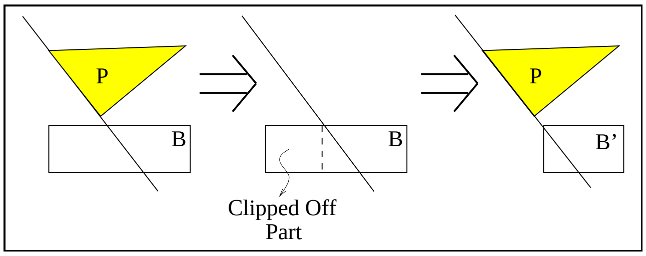

Figure 4: One "clip" of the clipping algorithm.

The clipping test is another algorithm designed to test the intersection between a hyper-rectangle and a set of linear constraints. This algorithm proceeds in rounds. The number of rounds used will be referred to as I. In every round, each constraint is applied once. During the application of a constraint, the constraint is used to "clip off" a piece of the hyper-rectangle B (see Figure 4).

The MBH of the clipped off shape is then used as a replacement for B (see Figure 4). If the entire box can be clipped away, the algorithm returns "No". If the box doesn't change after a round, the algorithm returns "Yes". If the number of rounds I has been reached, the algorithm returns "Yes". To illustrate the clipping algorithm, examine the query shape and box illustrated on the upper left corner of Figure 5. Figure 5 shows the results after each of the three constraints is applied. In this case, all of the original B was chopped away, so the algorithm returns "No". Only one round of clipping was necessary to determine the correct answer in the example.

1. $B := B \cap Bp$
2. **repeat** I times the following:
   a. **for** $i := 1$ **to** m **do**
      i. Find smallest possible B' s.t. $B' \subseteq B$ and $B \cap P_i \subseteq B'$.
      ii. **if** $B' = \emptyset$ **then return** "No".
      iii. $B := B'$.
   b. **if** B has not changed **then return** "Yes".
3. **return** "Yes"

Figure 6: The Clipping Algorithm.

The basic algorithm for the clipping test is given in Figure 6. Steps 2(a)(i),2(a)(ii) and 2(a)(iii) can be done as follows:

1. Define $\vec{z} = (z_1, .., z_n)$ as follows: **if** $A_{ij} > 0$ **then** $z_j = h_j$ **else** $z_j = l_j$
2. $d := c_i - \vec{A}_i \cdot \vec{z}$
3. **if** $d > 0$ **then return** "No"
4. **for** $j := 1$ **to** n **do** define l'_j, h'_j as follows:

   If $A_{ij} = 0$ **then** $l'_j := l_j$ **and** $h'_j := h_j$

   If $A_{ij} > 0$ **then** $h'_j := h_j$ **and** $l'_j := \max\{l_j, h_j + \frac{d}{A_{ij}}\}$

   If $A_{ij} < 0$ **then** $l'_j := l_j$ **and** $h'_j := \min\{h_j, l_j + \frac{d}{A_{ij}}\}$
5. $B := [l'_1, h'_1] \times \cdots \times [l'_n, h'_n]$

Note that the first three steps of the inner loop are the same steps used in the simple test. The box B can be clipped into $\emptyset$ if and only if $d > 0$. We have $d > 0$ if and only if some constraint P_i exists s.t. $B \cap P_i = \emptyset$. This is the criterion used in the simple test. Step 4 of the inner loop does the actual clipping. For each dimension j we try to clip B along that dimension using constraint P_i. If the coefficient for that dimension (A_{ij}) is 0, we can't clip using this dimension, so we set the boundaries of B' along dimension j to be exactly the same as B. If the coefficient is positive, we may be able to clip a part of B from the low side (i.e., make l'_j larger than l_j). Any $\vec{v} \in B$ that satisfies $v_j > \max\{l_j, h_j + \frac{c_i - \vec{A}_i \cdot \vec{z}}{A_{ij}}\}$ is outside P_i so we can safely raise l'_j to be $\max\{l_j, h_j + \frac{c_i - \vec{A}_i \cdot \vec{z}}{A_{ij}}\}$. A similar case occurs when the coefficient is negative and we lower h'_j.

Note also that the complexity of one full clip of B using a constraint P_i is $\mathcal{O}(n)$, and therefore the complexity of a full round of clipping is $\mathcal{O}(mn)$, which is linear in the size of the input.

Again the algorithm can be made safe by using a safe algorithm for calculating the dot product of two vectors. Using this algorithm for making the clipping algorithm safe is not a trivial task. For more details see Section 2.5 and Theorem 2.3.

The following claims can be made about the behavior of the clipping test:

Theorem 2.2 :

1. *The algorithm never yields a false negative.*
2. *For $n = 2$, if $P \neq \emptyset$ the algorithm reaches the correct result after one iteration (no false positives).*
3. *For $n = 2$, if no bound is placed on I, the algorithm always halts with the correct answer.*
4. *For any $n \geq 2$ and any $K > 0$, we can always find some input on which the algorithm stops after exactly $K+1$ iterations when no bound is placed on the number of iterations I.*
5. *For $n > 2$, there are cases when the algorithm stops with a false positive even when there is no bound on I.*
6. *Even when $I = 1$, this algorithm is strictly more accurate than the simple test.*
7. *Suppose $P_1, .., P_m$ are linear constraints and $Q_1, .., Q_m$ are the same constraints in a different order (i.e., $\{P_1, .., P_m\} = \{Q_1, .., Q_m\}$). Suppose B is some hyper-box. Suppose B' is the result of the clipping algorithm using $P_1, .., P_m$ on B with I iterations and B'' is the result of the clipping algorithm using $Q_1, .., Q_m$ on B with mI iterations. Then $B'' \subseteq B'$.*

Parts 2 and 3 of Theorem 2.2 show that in 2D, clipping is a very accurate and usually efficient test. Note that if Bp is tight (with respect to the query shape), clipping always terminates with the correct answer using one iteration. Part 4 of Theorem 2.2 proves that in general, no bound can be

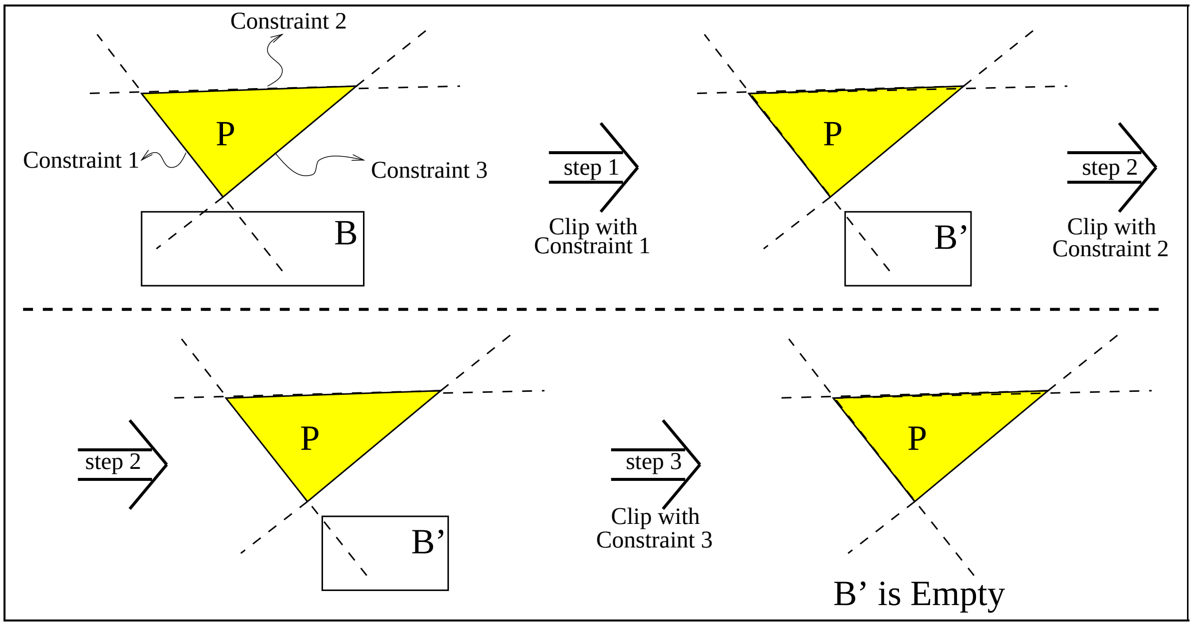

Figure 5: One iteration of the clipping algorithm

placed on the run time of the clipping algorithm when run to completion. Part 5 of Theorem 2.2 proves that even if the algorithm is run to completion, it is still conservative. Note that one round of the clipping test costs about twice as much as the entire simple test. However, using the clipping test can yield less false positive answers and hence less I/O for a query (e.g., see the results in Section 3.2). Part 7 of Theorem 2.2 shows that the order in which we apply constraints in the clipping algorithm has a limited effect on the accuracy of the result.

In summary, the clipping algorithm for testing overlap is usually more expensive and never less accurate than the simple test. In addition, the accuracy of the algorithm can be increased by spending more time in the test through increasing the number of iterations.

2.5 A Safe Algorithm for the Sum of n Numbers

In this section we introduce an algorithm for calculating the sum of n IEEE floating point numbers with minimal error. This section can be omitted without loss of continuity in the paper.

2.5.1 Problem Definition

Input: n IEEE floating point numbers (e.g., *floats, doubles, quads*) $x_1, .., x_n$.

Output: s such that $s = \sum_{i=1}^{n} x_i + \varepsilon$. ε is bounded by the magnitude of the last two bits in the mantissa of s. In other words, the result isn't off by more then the last two bits of the mantissa. In addition, s always has the correct sign. The only exception to these rules is when overflows occur, which we do not handle.

2.5.2 Some Intuition Concerning the Addition of Floating Point Numbers

The simplest algorithm for the addition of n numbers is as follows ($\oplus$ is IEEE standard floating point addition):

1. $x := x_1$
2. For $i := 2$ to n do $x := x \oplus x_i$
3. Return x

The result of the above algorithm may be arbitrarily far from the true result (assuming infinite precision arithmetic). For example, suppose the four numbers being summed are $-2^{1000}, 1, -1$ and 2^{1000}. Using different orders of summation,

$$((-2^{1000} \oplus 1) \oplus -1) \oplus 2^{1000} = 0$$
$$((-2^{1000} \oplus -1) \oplus 2^{1000}) \oplus 1 = 1$$
$$((-2^{1000} \oplus 1) \oplus 2^{1000}) \oplus -1 = -1$$

Clearly, the order of the various x_is matters. At first glance it seems that sorting the input numbers before doing the addition will help. Unfortunately, this is not always the case. Table 1 shows examples where sorting the input is inadequate.

2.5.3 Our Solution: Notation

The notation used in our algorithm includes several simple functions and a data structure. The functions are:

$\log n$. n is a positive integer. The result is $\lceil \log_2 n \rceil$.

$exp(x)$. x is an IEEE standard floating point number of the desired size and $exp(x)$ returns its exponent (as an integer).

$split(x, b)$. x is an IEEE standard floating point number of the desired size and b is a positive integer. We say that a number has b bits padding if the last b bits of its mantissa are 0. [2] The return value is two IEEE standard floating point numbers x_1, x_2 s.t. $x_1 + x_2 = x$

[2] We also say that there are b bits padding is the exponent is the minimal possible for that IEEE standard floating point size and if the first b bits of the mantissa are 0.

Sorting Order Description	Order in Math Symbols	Example	True Result
ascending	$\forall i < j : x_i < x_j$	$-2^{25} \oplus 1 \oplus 2^{25} = 0$	1
descending	$\forall i < j : x_i > x_j$	$2^{25} \oplus 1 \oplus -2^{25} = 0$	1
ascending (of absolute values)	$\forall i < j : \lvert x_i\rvert < \lvert x_j\rvert$	$1 \oplus 2^{25} \oplus -2^{25} = 0$	1
descending (of absolute values)	$\forall i < j : \lvert x_i\rvert > \lvert x_j\rvert$	$2^{25} \oplus \overbrace{1 \oplus \cdots \oplus 1}^{32 \text{ times}} = 33554432$	33554464

Table 1: Examples of addition with different sorting orders. Addition in the examples is done from left to right. Numbers are assumed to be of size *float*.

both numbers have b bits padding. If x has b bits padding then $x_1 = x$ and $x_2 = 0$. Obviously, this function can't work if b is too large (i.e. more than half the mantissa size).

$x \oplus y$. This is our notation for IEEE standard floating point addition. When we use the $+$ sign we refer to infinite precision addition.

Note that $\log n$, $exp(x)$ and $split(x, b)$ can be calculated very efficiently using bit-wise operations on integers. In addition to the above functions, we use a priority queue as a data structure. The items stored in the queue are IEEE standard floating point numbers where the priority is their exponents. The functions associated with a priority queue S are:

$|S|$. The size of the queue (how many numbers are in S).

$pop(S)$. Remove a number with maximal exponent from S and return it as the value.

$top(S)$. Return the value of a number with maximal exponent in S without changing S.

insert x into S. Put number x into S (using its exponent as priority).

2.5.4 Our Solution: Special Cases

We identify the solution to the summation problem for two special cases of input. In Section 2.5.5 we show how the general case can be reduced to these simpler cases. For both cases we assume that there are n input numbers. We define $k = \lceil \log_2 n \rceil$. We assume that all the input numbers have $2k$ bits padding.

Case 1. All the input numbers have the same sign. In this case, we put all the numbers in a priority queue S. The algorithm (called $sum_1(S)$) is described in Figure 7.

Case 2. There are both positive and negative numbers in the input. In this case, we split the input into two sets S_1, S_2 s.t. one set contains all the negative numbers and the other contains all the positive numbers. The condition is that $exp(top(S_1)) > exp(top(S_2)) + \log |S_2| + 1$. The (called $sum_2(S_1, S_2)$) is described in Figure 8.

The following theorems prove that the algorithms in the two special cases above meet our correctness requirements.

Lemma 2.1 *: Suppose S is an input satisfying the conditions of case 1. Suppose $\varepsilon = \sum S - sum_1(S)$ and m is the size of mantissa used for sum_1. Then, $exp(\varepsilon) \le exp(sum_1(S)) - m$. (We assume the exponent of 0 is $-\infty$.)*

Lemma 2.2 *: Suppose S_1, S_2 is an input satisfying the conditions of case 2. Suppose $\varepsilon = \sum(S_1 \cup S_2) - sum_2(S_1, S_2)$ and m as before. Then, $exp(\varepsilon) \le exp(sum_2(S_1, S_2)) - m$.*

1. **if** $S = \emptyset$ **then return** 0
2. $x := pop(S)$
3. $e := exp(x)$
4. **while** $S \neq \emptyset$ **and** $exp(top(S)) \ge e - k$ **do**
   $x := x \oplus pop(S)$
5. $x' := sum_1(S)$
6. **return** $x \oplus x'$

Figure 7: Algorithm $sum_1(S)$.

1. $x := pop(S_1)$
2. $e := exp(x)$
3. **while** $S_1 \neq \emptyset$ **and** $exp(top(S_1)) \ge e - k$ **do**
   $x := x \oplus pop(S_1)$
4. $x' := sum_1(S_1)$
5. $y := sum_1(S_2)$
6. **return** $x \oplus (x' \oplus y)$

Figure 8: Algorithm $sum_2(S_1, S_2)$.

2.5.5 Our Solution: The General Case

We now show how the general case can be reduced to the simpler cases presented in Section 2.5.4. The input $\mathcal{S}$ contains n IEEE standard floating point numbers (of equal size). We have a preprocessing stage in which we put the numbers of $\mathcal{S}$ into two priority queues, one for positive numbers and the other for negative numbers. We also ensure some necessary padding conditions. After preprocessing we perform the main part of the algorithm. Informally, the following summarizes the main algorithm:

Iterate until one of the sum routines is called :

1. Check if input conforms to case 1. If it does, use sum_1 to get the result.
2. Check if input conforms to case 2. If it does, use sum_2 to get the result.
3. Transform the input into a form closer to one of the special cases.

The complete algorithm is in Figure 9. The preprocessing part creates sets S_1 and S_2 such that S_1 contains all the positive numbers and S_2 all the negative numbers. The preprocessing also ensures that the $2k$ padding requirement for the special cases is met. This is done by using *split* when necessary. Note that the definition of k is one larger than the stated value in sum_1, sum_2. This is done since $|\mathcal{S}|$ can double if every number in it splits.

Step 1 of the main part checks if the condition for sum_1 is satisfied. If so, then sum_1 is used to get the right answer.

Step 2 makes sure that the largest exponent of S_1 is greater or equal to the largest exponent of S_2. If the condi-

Preprocessing. Do the following steps:

1. $k := 1 + \log|\mathcal{S}|$
2. Set S_1, S_2 as empty queues.
3. **for each** $x \in \mathcal{S}$ **do**
   a. $\{x_1, x_2\} := split(x, 2k)$
   b. **if** $x_1 > 0$ **then**
      i. insert x_1 into S_1
      ii. **if** $x_2 \neq 0$ **then** insert x_2 into S_1
   c. **if** $x_1 < 0$ **then**
      i. insert x_1 into S_2
      ii.**if** $x_2 \neq 0$ **then** insert x_2 into S_2

Main part. Iterate until one of the sum routines is called:

1. **if** $S_1 = \emptyset$ **or** $S_2 = \emptyset$ **then return** $sum_1(S_1 \cup S_2)$
2. **if** $exp(top(S_1)) < exp(top(S_2))$ **then** $switch(S_1, S_2)$
3. **if** $exp(top(S_1)) - exp(top(S_2)) > 1 + \log|S_2|$ **then return** $sum_2(S_1, S_2)$
4. $x := pop(S_2)$
5. **while** $exp(x) < exp(top(S_1))$ **and** $S_2 \neq \emptyset$ **and** $exp(top(S_1)) - exp(top(S_2)) \geq 2k$ **do**
   $x := x \oplus pop(S_2)$
6. $x := x \oplus pop(S_1)$
7. $\{x_1, x_2\} := split(x, 2k)$
   For the next two steps we assume S_i is the set with numbers of the same sign as x
8. **if** $x_1 \neq 0$ **then** insert x_1 into S_i.
9. **if** $x_2 \neq 0$ **then** insert x_2 into S_i.

Figure 9: Algorithm for sum of n numbers.

tion is not satisfied then the sets are switched. This ensures the relative exponent sizes of S_1 and S_2.

Step 3 Checks if the condition for sum_2 is satisfied. If so, then sum_2 is used to get the correct answer.

Steps 4-9 reduces the problem size by performing additions without numerical error. In Steps 4 and 5 we accumulate some numbers from S_2 until either the accumulator x has the same exponent as the largest exponent in S_1 or we run out of numbers. Then, in Step 6 we add x to a number with the largest exponent in S_1. Note that this addition is done without rounding error. In Steps 7-9 we put the result back into the proper set. We may use *split* to make sure that the padding requirement is satisfied.

The following lemma are needed to prove that the algorithm is correct and has the required error bounds:

Lemma 2.3 *:*

1. *There is no rounding error in Steps 4-9. (i.e., the additions done in Steps 5,6 are without rounding errors.)*
2. *After going through Steps 4-9 one of the following happens:*
   (a) $|S_1 \cup S_2|$ *is reduced.*
   (b) The condition for using sum_1 *is satisfied.*
   (c) The condition for using sum_2 *is satisfied.*

The following Theorem states the correctness of the algorithm presented in Figure 9:

Theorem 2.3 :

1. *The algorithm performs* $\mathcal{O}(n)$ *floating point operations.*
2. *The algorithm performs* $\mathcal{O}(n \log n)$ *integer operations.*
3. *The error of the result is bounded by the magnitude of the last two bits of the mantissa of the result.*

Lemma 2.3 shows that each iteration either reduces the number of numbers in S_1, S_2 without any error, or reduces S_1, S_2 to a state where we can use sum_1 or sum_2. This means that the algorithm stops eventually by using sum_1 or sum_2 so the error in the algorithm is not larger than the required error as proven for sum_1 and sum_2. Using Lemmas 2.1 and 2.2 we see that the error of the result is bounded by the magnitude of the last two bits of the mantissa of the result. This also means that the sign of the result is with no error. The worst case for this algorithm is when all the numbers in $\mathcal{S}$ need to be split in the preprocessing and then every iteration (except the last one) reduces $|S_1 \cup S_2|$ by 1. In this case we have at most $4|\mathcal{S}|$ additions before we get the result. We also perform $\mathcal{O}(n)$ operations on priority queues. Each such operation costs $\mathcal{O}(\log n)$ so in all we perform $\mathcal{O}(n \log n)$ integer operations.

3 Performance Evaluation

This section describes some experiments we performed with our new processing strategy for QBLC. We present here only a summary of the main results. For more detailed description of the experiments, including an experiment with a real high dimensional business data set, see [5].

Throughout this study, performance results of finite area QBLC queries are compared to performance results of the corresponding MBH query, based on a careful implementation of R-Trees with the search algorithms. We report elapsed time for results gathered using the Shore storage manager, and I/O for results gathered using Unix files. We refer to the MBH query as STD (standard) in the rest of this section. We call it "standard" because using a bounding box approximation to the query shape is the best processing strategy previously suggested for such queries.

In our experiments we systematically varied the following query parameters: **selectivity**, **shape** (ratio of query area over bounding box area) and **complexity** (number of constraints used for specifying the query). We also varied the data sets. In particular, we varied the amount of **overlap** in the data set and the **dimensionality**. Finally, we also varied the **page size** and **buffer pool size** in the system. For sake of brevity, we do not show here the results for data overlap, page size and buffer pool size. The effect of data overlap and page size was minor. The effect of buffer pool size was negligible. We refer to our processing strategy (using the clipping algorithm with one round of clipping) as "NEW".

3.1 Experiments with the Tiger Data Set

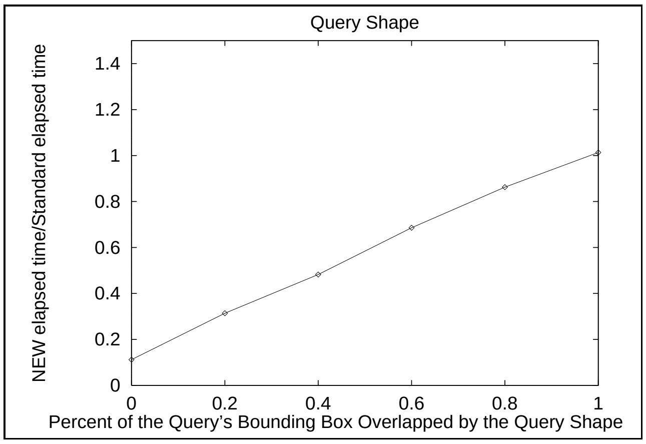

Figure 10: Varying Query Shape

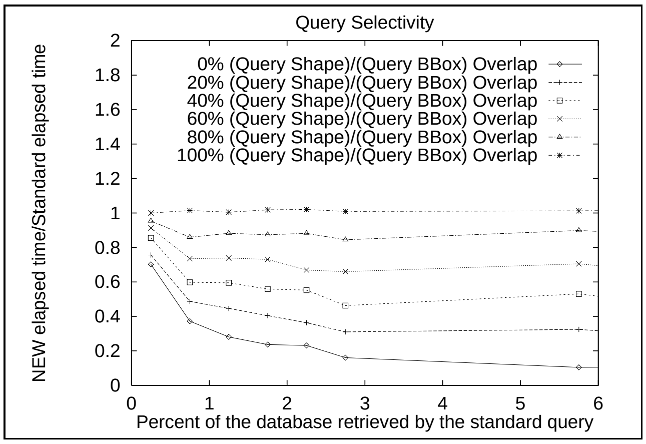

Figure 11: Varying Selectivity

Figure 10 shows the effect of query shape on performance. This experiment used the TIGER/Line data set. The data set contained 150000 tuples. Each tuple is a line segment represented by four integers. The size of the data set is 2.4MB. Note that we vary the ratio (query shape area)/(area of query MBH) between 0 and 1. Geometrically, the query shapes for these two extremes are a diagonal line and a box respectively. Obviously, there is no advantage to using polygons to bound a query that is a box, and there is maximal advantage to doing so when the query is a diagonal line. While we varied the query shape, we kept the size of the MBH within a given range, so that the standard algorithm for each query retrieved five to ten percent of the objects in the database. The query to MBH overlap ratio roughly equals the improvement achieved by the constraint search algorithm; note that that at no point does the cost of the constraint algorithm exceed the cost of the standard search algorithm. This is because the overhead of polygon overlap tests is very small for queries in which the bounding polygon has only a few points.

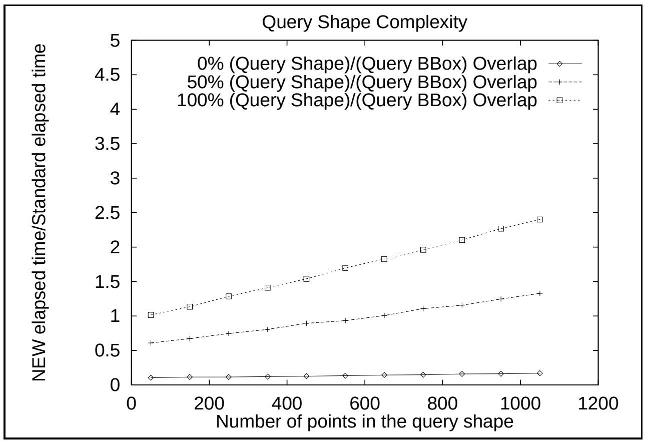

Figure 12: Varying Query Complexity

Figure 11 has multiple lines. Each line represents the effect of selectivity on a particular query shape. Small queries are similar in size to individual objects in the database, and all answers are likely to be clustered close together. As a result, there is less to be gained from a more accurate representation of such queries. This explains the similarity in performance between the different query shapes for small selectivities as well as the query shapes' subsequent divergence. The worst case for the constraint algorithm is obviously when the query is itself a box (100% overlap); note that even here, the constraint algorithm costs essentially the same as the standard algorithm regardless of the query shape. This is an important observation: *while the benefits of using our algorithm can be great (an order of magnitude improvement), even in the worst case it is only marginally slower than STD.*

Figure 12 shows the effect of varying the query complexity. The complexity of a query, in terms of the number of points in the bounding polygon, affects the cost of the constraint overlap test. This experiment uses the same data set as the one that varied query selectivity, but generates queries a little differently. The same basic query shape is still used, but the number of edges in the bounding polygon is artificially increased by using redundant constraints. Among all our experiments, this is the only one in which the constraint algorithms were seen to perform worse than the standard algorithm. This experiment also indicates that the constraint algorithm is superior as long as the number of points is kept under some maximum. This maximum is on the order of 500 constraints which is far more than we expect to see in a normal query.

3.2 Experiments with a Synthetic Five Dimensional (5D) Data Set

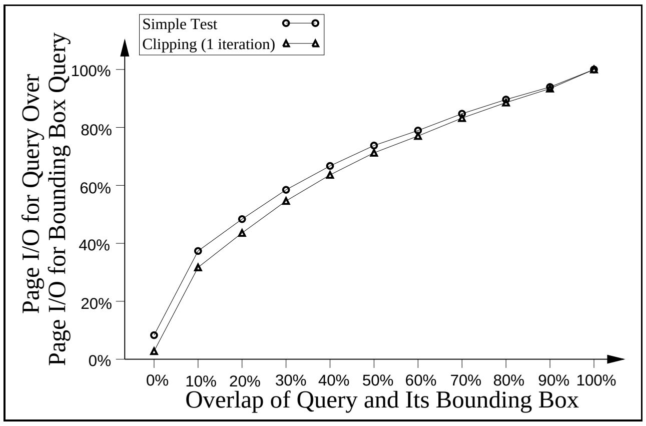

Figure 13: 5D data set. Bounding box size is 1% of the data space.

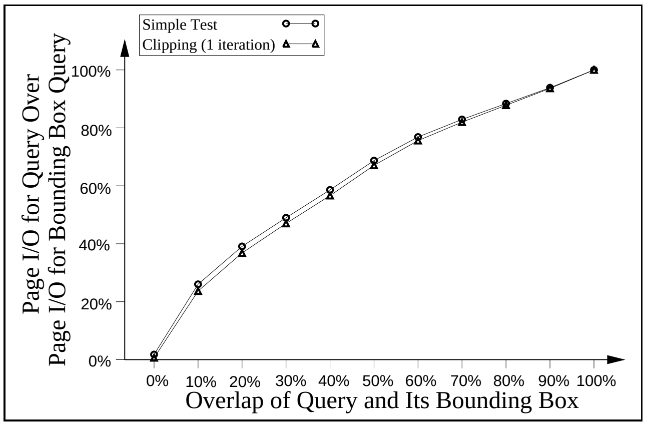

Figure 14: 5D data set. Bounding box size is 10% of the data space.

The data set used in this experiment contained 6 million five dimensional tuples (the data set size is 120MB), where the dimensions were independent and uniformly distributed. This artificial data set was bulk loaded using the STR bulk loading algorithm [10]. The experiments were of the same variety as the 2D experiments. We examined the ratio of I/O costs incurred by running linear constraint queries and their corresponding bounding hyper-rectangles. We varied the ratio of hyper-volumes between the linear constraint queries and their hyper-rectangles. Note that the bounding box B_p was not used in these experiments even though it was known. This is more realistic since high-dimensional queries are likely to be algebraic and no B_p will be available.

Figure 13 shows the results when the linear constraints' bounding boxes overlapped 1 percent of the data set. Figure 14 shows the results when the linear constraints' bounding boxes overlapped 10 percent of the data set. Note the similarity of behavior to the 2D experiments. Also, note from Table 2 the improvement in accuracy achieved by using clipping over the simple test. Note that the improvement is most noticeable for the query shapes whose hyper-volume is small compared to their bounding boxes' query shape.

These experiments demonstrate the effectiveness of our overlap tests for data of dimensionality higher than two. In addition, they show the usefulness of the clipping test over the simple test for these situations.

4 Application of Our Tests to Other Indexing Structures

The tests we've developed, which determine whether a hyper-rectangle and conjunction of linear constraints overlap, can be used to process QBLC in a wide variety of indexing structures. For instance, there are indexing structures like R-Trees [7], Buddy Trees [9] , k-d-B Trees [14], and Grid Files [12], which either implicitly or explicitly use hyper-rectangles as their means of spatial subdivision.

For example, our tests can also be used search k-d-B Trees. A k-d-B tree is a binary tree where each node splits some dimension along some value for that dimension. In this case, if we start with hyper-rectangular bounds on the data base (these bounds can be conservative), each node of the k-d-B tree splits the hyper-rectangular region represented by the node into two hyper-rectangular sub regions. Thus, when performing QBLC, each of the sub-regions can be tested against the query using our techniques. This allows the search to be more intelligently directed, just as in R*-Trees using QBLC.

As another example consider how our tests can be used in Grid Files. Before a pointer in a catalog entry is used to fetch a data page, one can find the bounding box of the catalog entry from the linear scales of the Grid File. (Incidentally, this box "contains" all the data points on the page it points to.) This box can be tested against the query, and the page is fetched only if the box overlaps the query region.

5 Conclusions and Future Work

This paper studies a useful query form that is currently poorly optimized but is expressible in SQL. Techniques for using existing indexing structures for QBLC optimization are discussed, and two new tests are presented for QBLC optimization in indexing structures that use hyper-rectangles to subdivide the data space. Both of these tests are efficient, and guarantee that no answers are lost due to numerical error.

The performance studies clearly demonstrate the value of QBLC optimization for common use in a variety of domains. Included in these domains are GIS and OLAP. In particular, even the worst case performance was no worse than current solutions, which approximate QBLC queries with MBHs.

Further extensions to this work include studying QBLC optimization with other indexing structures, exploring the optimization of higher-order polynomial constraints, and studying QBLC optimization in other problem domains.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1992.

[2] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, N.J., 1963.

[3] D. DeWitt, N. Kabra, J. Luo, J. Patel, and J. Yu. Client-server paradise. In *Proc. 20th Int. Conf. on VLDB*, pages 558–569, 1994.

Overlap of query and its bounding box	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Ratio of clipping test I/O over simple test I/O	33.5%	84.8%	90.1%	93.4%	95.4%	96.6%	97.6%	98.2%	98.8%	99.4%	100%

Table 2: Ratio of I/O for clipping test vs. simple test. Same data set used for Figure 13.

[4] S. C. Fang and S. Puthenpura. *Linear Optimization and Extensions: Theory and Algorithms.* Prentice Hall, 1993.

[5] J. Goldstein, R. Ramakrishnan, and U. Shaft abd J.-B. Yu. Using constraints to query R*-Trees. Technical Report TR1301, Comp. Sci. Dept., Univ. of Wisconsin-Madison, 1996.

[6] O. Günther and E. Wong. A dual approach to detect polyhedral intersection in arbitrary dimensions. In *Proc. 25th Annual Allerton Conf. on Comm., Control and Comp.*, pages 859–868, 1987.

[7] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–57, 1984.

[8] H.-P. Kriegel, M. Schiwietz, R. Schneider, and B. Seeger. Performance comparison of point and spatial access methods. In *SSD*, pages 89–113, 1989.

[9] H.-P. Kriegel, M. Schiwietz, R. Schneider, and B. Seeger. The buddy-tree: An efficient and robust method for spatial data base systems. In *Proc. 16th VLDB Conf.*, pages 590–601, 1990.

[10] Scott T. Leutenegger, Jeffrey M. Edington, and Mario A. Lopez. STR: A simple and efficient algorithm for R-Tree packing. Technical Report 96-02, Mathematics and Computer Science Dept., University of Denver, 1996.

[11] Ketan Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice-Hall, Simon and Schuster Company at Englewood Cliffs, NJ, 1994.

[12] J. Nievergelt, H. Hinterberger, and S. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. In *Readings in Database Systems.* Morgan Kaufmann, 1988.

[13] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1993. 3rd printing.

[14] J. Robinson. The k-d-b tree: A search structure for large multi-dimensional dynamic indexes. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 10–18, 1981.

[15] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-Tree: A dynamic index for multi-dimensional objects. In *Proc. 13th Inf. Conf. on VLDB*, pages 507–518, 1987.

A The Simple Test in Detail

The simple test is based on the following observation:
If $\exists i$ s.t. $B \cap P_i = \emptyset$ then $B \cap P = \emptyset$.
Proof Note that $P = \bigcap_{i=1}^{m} P_i$.
Therefore, $B \cap P = \bigcap_{j=1}^{m} (B \cap P_j)$.
This means that:
$B \cap P_i = \emptyset \implies \bigcap_{i=1}^{m} (B \cap P_i) = \emptyset \implies B \cap P = \emptyset$.
■
The simple test is:

1. For $i = 1$ to m do
   If $B \cap P_i = \emptyset$ return "No"
2. Return "Yes"

Assuming that we can accurately find whether $B \cap P_i = \emptyset$, this algorithm is obviously conservative. How do we know if $B \cap P_i = \emptyset$? If exists $\vec{x} \in B$ s.t. $\vec{A}_i \cdot \vec{x} \geq c_i$ then $B \cap P_i \neq \emptyset$, otherwise $B \cap P_i = \emptyset$. Define $\vec{z} = (z_1, .., z_n)$ as follows: If $A_{ij} > 0$ then $z_j = h_j$. Otherwise $z_j = l_j$. Clearly we have that

$$\max\{\vec{A}_i \cdot \vec{x} | \vec{x} \in B\} = \vec{A}_i \cdot \vec{z}$$

This means that $B \cap P_i = \emptyset$ if and only if $\vec{A}_i \cdot \vec{z} \geq c_i$.

B The Simple Test: Theoretical Results

Part 1 of Theorem 1: *Both the simple test and the refined simple test never yield a false negative. Furthermore, both tests cost $\mathcal{O}(mn)$ (which is linear in the input).*
Proof The description of the algorithm in Appendix A (with the safe dot product algorithm) shows that both versions of the simple test never yield a false negative. Remember that the input contains $m(n+1) + 2n$ numbers (m constraints, each represented by $n+1$ numbers, and a box represented by $2n$ numbers). This means that the input has size $\mathcal{O}(mn)$. From the safe dot product algorithm we know that the cost of a dot product is linear in the input. The simple test has an outer loop that is done m times. Each time we do a dot product of vectors of size $n+1$ which costs $\mathcal{O}(n)$. Therefore, the total cost of the simple test is $\mathcal{O}(mn)$. The refined simple test has the additional cost of intersecting B with B_P. This costs $\mathcal{O}(n)$, so the entire refined simple test costs $\mathcal{O}(mn)$. ■
Part 2 of Theorem 1: *For two dimensions ($n = 2$), if B_P is a minimal box that contains P, the refined simple test never yields a false positive.*
Proof suppose that B_P overlaps B and that P does not overlap B. We will show that there is a linear constraint in P such that no vertex of B satisfies it. (This would mean that the simple test returns the answer "No".)

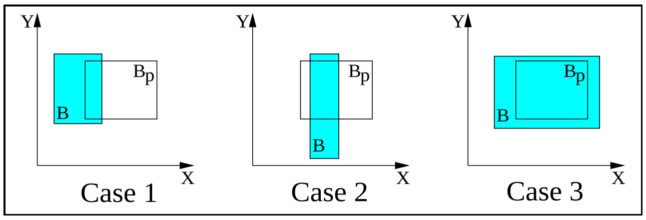

Figure 15: Intersection of B and B_P where B doesn't have a vertex inside B_P.

First, we want to see that exists a vertex of B inside B_P. Figure 15 shows some cases of B intersecting B_P where no vertex of B is in B_P. In Cases 1 and 3 (of the figure) B contains at least one entire edge of B_P. Since $B \cap P = \emptyset$, we have that P doesn't touch this edge of B_P. This is clearly a contradiction to B_P being a tight bounding box of P. In Case 2 (of the figure) B_P is partitioned into three parts (left, middle and right parts). P can reside only in the left part or the right part but not in both. In any case, it follows that B_P is not a tight bounding box of P, which is the same contradiction we got before. Any other case of B intersecting B_P without having a vertex of B in B_P is a rotated or reflected version of one of the three cases. Therefore, B has a vertex $p = (x_p, y_p)$ s.t. $p \in B_P$.

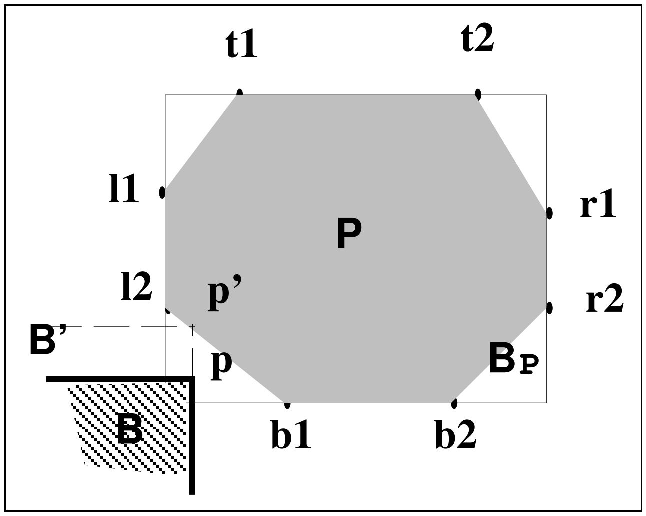

Figure 16: Illustration of Proof

Since B_P is a minimal bounding rectangle, P must intersect each of B_P's edges. Consider the side of B_P that is parallel to the X axis and farthest from it. Without loss of generality, let $t1$ be the point on this side of B_P that intersects with P, and of all such intersection points on this side, is the closest to the Y axis. This is illustrated in Figure 16. The points $t2$, $r1$, $r2$, $b1$, $b2$, $l1$ and $l2$ are similarly defined. (Note that the points are not necessarily distinct; for example, if P intersects the top of B_P at only one point, $t1$ and $t2$ are the same. Similarly, if P happens to be a diagonal line from the lower left to the top right corner, $t1$, $t2$, $r1$ and $r2$ are the same, and $b1$, $b2$, $l1$ and $l2$ are the same.)

Since we have assumed that B and P do not overlap, the point p must be in one of the unshaded corner regions. Without loss of generality, let this be the lower left region. If some vertex of B is in this region, and the top right vertex of B is not in this region, then B and B_P necessarily overlap. By our hypothesis, P does not overlap B. Thus, the top right vertex of B is also in the unshaded lower left region of B_P. Without loss of generality, let the point p be the top right vertex of B. (In this case, of course, the points $l2$ and $b1$ must be distinct.)

We can expand B upwards (increasing y_p while keeping x_p constant) such that the expanded B overlaps P. Let $\varepsilon > 0$ be minimal such that the box B' (obtained by expanding B upward) with top right corner $p' = (x_p, y_p + \varepsilon)$ intersects P; clearly, the point p' must lie on the boundary of polygon P, by virtue of the minimality of ε.

Without loss of generality, we will assume that p' is a vertex of P. (If it is not a vertex of P we can define a dummy vertex of P by "splitting" the edge it is on into two edges that are defined by the same linear constraint.) Let $a_1 Y + b_1 X + c_1 \geq 0$ and $a_2 Y + b_2 X + c_2 \geq 0$ be the two constraints of P that define the edges intersecting at p'. (If we define a dummy vertex of P then those two constraints are the same.) Since both B' and P are convex polygons there is a small circular environment of p', which we will call C, such that a point $q = (x_q, y_q) \in C$ belongs to P if and only if it satisfies both constraints, and $q \in B'$ if and only if $x_q \leq x_p$ and $y_q \leq y_p + \varepsilon$. Furthermore we have $a_i x_p + b_i(y_p + \varepsilon) + c_i = 0$.

We claim that one of the two constraints of P that intersect at p' is such that no vertex of B satisfies it. The intuition is that at least one of these constraints has a negative slope ($\partial x / \partial y$) and excludes the lower half-plane (which contains the vertexes of B). The claim is proven below by a case analysis, completing the proof of the theorem:

1. $a_i > 0$ **and** $b_i \geq 0$ **for either** $i = 1$ **or** $i = 2$ **:** Then for any vertex (x, y) of B, $a_i y + b_i x + c_i < a_i x_p + b_i(y_p + \varepsilon) + c_i = 0$ holds. So none of the vertices of B satisfy the constraint and the theorem is proved.

2. $a_1 \leq 0$ **and** $a_2 \leq 0$ **:** In this case, consider a point in C of the form $q = (x_p, y_p + \varepsilon - \delta)$ for $\delta > 0$. q is clearly on B' and satisfies $a_i(y_p + \varepsilon - \delta) + b_i x_p + c_i = -a_i \delta \geq 0$. This contradicts the fact that $B' \cap P = \{p\}$. Therefore, this case cannot arise.

3. $b_1 \leq 0$ **and** $b_2 \leq 0$ **:** As in Case 2, we can examine $q = (x_p - \delta, y_p + \varepsilon)$ and obtain the same contradiction.

4. $a_1 \leq 0$, $b_1 > 0$, $a_2 > 0$ **and** $b_2 < 0$ **:** There are two subcases to consider.

4a. $a_2 b_1 \geq a_1 b_2$ **:** Consider a point in C represented by $q = (x_p - \delta a_1, y_p + \varepsilon + \delta b_1)$. This point is in B' and it satisfies both constraints:

$$a_1(y_p + \varepsilon + \delta b_1) + b_1(x_p - \delta a_1) + c_1 =$$
$$[a_1 x_p + b_1(y_p + \varepsilon) + c_1] + \delta[a_1 b_1 - b_1 a_1] = 0$$
$$a_2(y_p + \varepsilon + \delta b_1) + b_2(x_p - \delta a_1) + c_2 =$$
$$[a_2 x_p + b_2(y_p + \varepsilon) + c_2] + \delta[b_1 a_2 - b_2 a_1] \geq 0$$

So we obtain a contradiction; this case cannot arise.

4b. $a_2 b_1 < a_1 b_2$ **:** In this case, suppose that $q = (x_p + a, y_p + \varepsilon + b)$ is any point that satisfies both constraints. Then we have

$$a_1(y_p + \varepsilon + b) + b_1(x_p + a) + c_1 \geq 0 \qquad \Longrightarrow$$
$$-b_2 a_1 b - b_2 b_1 a \geq 0$$
$$a_2(y_p + \varepsilon + b) + b_2(x_p + a) + c_2 \geq 0 \qquad \Longrightarrow$$
$$b_1 a_2 b + b_1 b_2 a \geq 0$$

If we sum them both we see that $b[a_2 b_1 - a_1 b_2] \geq 0$. Since $a_2 b_1 - a_1 b_2 > 0$ we have $b \geq 0$, so $y_p + \varepsilon$ is a minimal value for points in P and hence there is no intersection between B and B_P. This is a contradiction, so this case cannot arise.

5. $a_2 \leq 0$, $b_2 > 0$, $a_1 > 0$ **and** $b_1 < 0$ **:** This case is the same as Case 4, differing only in the order of constraints.

This completes the proof of the theorem. ■

Figure 17 shows an example for $n = 2$ where if B_P is not given then the algorithm answers with a false positive. The problem can be written as:

$$P \equiv \begin{cases} -x_1 + x_2 & \geq & 0 \\ -x_2 & \geq & -4 \\ x_1 + x_2 & \geq & 4 \end{cases} \qquad B \equiv [0,4] \times [0,1]$$

Figure 18 shows an example for $n = 3$ where the algorithm answers with a false positive even when a tight bounding box B_P is given. The problem can be written as:

$$P \equiv \begin{cases} -x_2 & \geq & -4 \\ x_1 & \geq & 0 \\ -x_3 & \geq & -4 \\ -x_1 + x_3 & \geq & 0 \\ x_1 + x_2 + x_3 & \geq & 4 \\ -x_1 + x_2 - x_3 & \geq & -4 \end{cases} \qquad B \equiv [3,4] \times [0,1] \times [0,4]$$

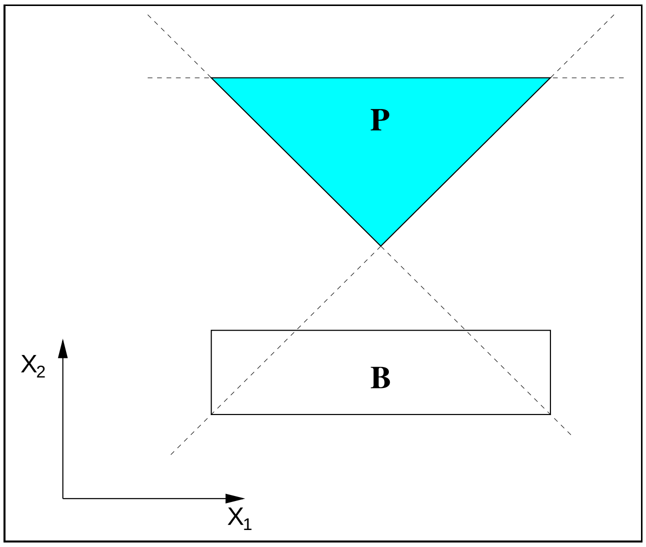

Figure 17: Example in 2D where the claim doesn't hold if B_P is not given.

Part 3 of Theorem 1: *If B is a point (i.e., $\forall i \quad l_i = h_i$), the simple test never yields a false positive.*
Proof Since B is a point we can denote it by $B = \{\vec{p}\}$. Since $P = \bigcap_{i=1}^{m} P_i$ we have

$$\vec{p} \in P \iff \vec{p} \in \bigcap_{i=1}^{m} P_i \iff \forall i: \quad \vec{p} \in P_i$$

The loop in the simple test checks whether a vertex of B is inside P_i for all P_i. In this case, $\vec{p}$ is the only vertex of B. Therefore, the simple test checks for every P_i whether $\vec{p} \in P_i$. Therefore, the simple test returns "Yes" if and only if $\forall i: \quad \vec{p} \in P_i$. Therefore, the simple test returns "Yes" if and only if $\vec{p} \in P$. ■

C The Clipping Test in Detail

The intuition for the Clipping algorithm comes from the following result:
Proposition: If $B' \subseteq B$ and $\exists i$ s.t. $B \cap P_i \subseteq B'$ then

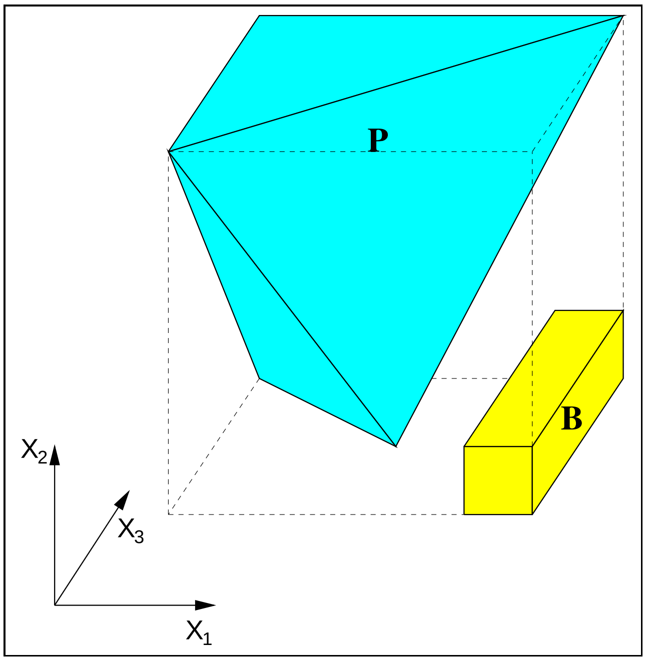

Figure 18: Example in 3D where the claim doesn't hold.

$B \cap P = B' \cap P$.

Proof Let B' and i be s.t. $B' \subseteq B$ and $B \cap P_i \subseteq B'$. Since $B' \subseteq B$ we have $B' \cap P \subseteq B \cap P$. Since $B \cap P_i \subseteq B'$, we have $B \cap P_i = (B \cap P_i) \cap P_i \subseteq B' \cap P_i$. Therefore, $B \cap P = (B \cap P_i) \cap P \subseteq (B' \cap P_i) \cap P = B' \cap P$. We have inclusion in both sides so: $B \cap P = B' \cap P$. ∎

This result means that when can look at a specific constraint (say P_i) and find some $B' \subseteq B$ s.t. $B \cap P_i \subseteq B'$. For such a B', $B' \cap P = B \cap P$ holds, and so we can safely replace B with B'. If we also have $B' \neq B$ (i.e. $B' \subset B$) then we have safely "clipped" the query box into a smaller box.

The basic step of the Clipping algorithm consists of taking a constraint P_i and box B and finding the smallest possible B' s.t. $B' \subseteq B$ and $B \cap P_i \subseteq B'$; the box B is then replaced by B'. By "smallest possible" we do not mean the actual smallest B', but the smallest B' we can find using safe numeric calculations. An *iteration* of the Clipping algorithm consists of applying this step to each constraint, considering the constraints in a round-robin order. After each such application, we check to see if the resulting B is empty or not. If it is empty then we can safely stop with a "No" answer. If it is not empty we continue.

The Clipping algorithm halts if there is no change in box B during an iteration. We also stop the algorithm after a predetermined number of iterations even if the last iteration has produced some change in B. We denote this predetermined number of iterations as I. An exact form of the algorithm is as follows: The general form of the algorithm is:

1. Repeat I times the following:
   (a) For $i := 1$ to m do
      i. Find smallest possible B' s.t. $B' \subseteq B$ and $B \cap P_i \subseteq B'$.
      ii. If $B' = \emptyset$ stop with output "No".
      iii. Set $B := B'$.
   (b) If B has not changed then stop with output "Yes".
2. Return "Yes"

The only detail left is how to find B' for a specific P_i. First we look at "clipping" B along one dimension j. This means that we try to change the values of l_j, h_j while keeping the values of the other dimensions the same. There are three different cases to consider:

If $A_{ij} > 0$: Define a vector $\vec{z} = (z_1, .., z_n)$ as follows: If $A_{ik} > 0$ then $z_k = h_k$ otherwise $z_k = l_k$. Let $a \in \Re$ be such that $A_{ij}a + \sum_{k \neq j} A_{ik}z_k = c_i$. We have that

$$a = h_j + \frac{c_i - \vec{A}_i \cdot \vec{z}}{A_{ij}}$$

For every $\vec{x} \in B$ s.t. $x_j < a$ we have that $\vec{A}_i \cdot \vec{x} < c_i$ so $\vec{x} \notin P_i$. This means that B' will differ from B only in its low value for dimension j, which will be $\max\{l_j, a\}$. For safety purposes we make every possible numerical error in the calculation of a on the low side so the safe value of a may eventually be lower than the optimal value but not higher. If we deduce that $a > h_j$, then the resulting B' is $\emptyset$.

If $A_{ij} < 0$: Define the vector $\vec{z}$ to be the same as in the previous case. Also find the same value for a as in the previous case but using

$$a = l_j + \frac{c_i - \vec{A}_i \cdot \vec{z}}{A_{ij}}$$

Note that for every $\vec{x} \in B$ s.t. $x_j > a$ we have that $\vec{A}_i \cdot \vec{x} < c_i$ so $\vec{x} \notin P_i$. This means that B' will differ from B only in its higher value for dimension j, which will be $\min\{h_j, a\}$. Note that this time we safely estimate a on the higher side to get a safe "clip". If we get $a < l_j$ then the result is $B' = \emptyset$.

If $A_{ij} = 0$: There is no way to clip B in dimension j using P_i.

Using these observations we can compute B' by going through the calculations and clipping B for each dimension. A step (clipping by constraint i and all dimensions) can be done as follows:

1. Define $\vec{z} = (z_1, .., z_n)$ as follows: If $A_{ij} > 0$ then $z_j = h_j$ else $z_j = l_j$.
2. Define $d = c_i - \vec{A}_i \cdot \vec{z}$.
3. For each $1 \leq j \leq n$ define l'_j, h'_j as follows:

   If $A_{ij} = 0$ then $l'_j = l_j$ and $h'_j = h_j$

   If $A_{ij} > 0$ then $h'_j = h_j$ and $l'_j = \max\{l_j, h_j + \frac{d}{A_{ij}}\}$

   If $A_{ij} < 0$ then $l'_j = l_j$ and $h'_j = \min\{h_j, l_j + \frac{d}{A_{ij}}\}$
4. Set $B' = [l'_1, h'_1] \times \cdots \times [l'_n, h'_n]$

Note that this calculation of B' may yield an empty box. This happens if and only if $d > 0$. To save some of the cost of the algorithm in this case we can insert a step "If

$d > 0$ return "No intersection"" between steps 2 and 3 of the algorithm.

Note that the only time when a clip of one dimension affects a clip of another dimension is when the result is $\emptyset$. The result for B' does not change when we change the order of dimensions in any way. The complexity of this method for the basic step of clipping a box with a constraint is $\mathcal{O}(n^2)$, and therefore the complexity of an iteration is $\mathcal{O}(mn^2)$. We observe that $\vec{z}$ remains the same when we deal with different dimensions, and the real cost is re-calculating $c_i - \vec{A}_i \cdot \vec{z}$ repeatedly. If we do this calculation once and use the result for each dimension, the cost for a basic clipping step is $\mathcal{O}(n)$, and the cost for an iteration is $\mathcal{O}(mn)$.

The reason we use a predetermined number of iterations as an additional stopping criterion is that the number of iterations is not bounded. The following input demonstrates the problem:

$$P \equiv \begin{cases} x_1 + x_2 & \geq & \varepsilon \\ -x_1 - x_2 & \geq & \varepsilon \\ -x_1 + x_2 & \geq & \varepsilon \\ x_1 - x_2 & \geq & \varepsilon \end{cases} \qquad B \equiv [-4,4] \times [-4,4]$$

For $\varepsilon > 0$ we have that $P = \emptyset$, and the number of iterations is at least $1/\varepsilon$. A similar example exists in 3 dimensions where we have $1/\varepsilon$ iterations and $P \neq \emptyset$. These examples show us that we can't afford to wait until the algorithm finishes all the iterations. If we stop the algorithm after as many iterations as we want we get a conservative algorithm.

We currently do not know if there is a situation for which the algorithm does not stop at all when I is infinite. An interesting observation is that the safe version of the algorithm does always stop but remains conservative. The reason is that after each iteration, B is still represented by IEEE standard floating point double precision numbers so the "clippings" occur in a discrete space, and thus the number of iterations is finite. This result does not help much in practice since the number of iterations can still be too large.

D Some Properties of the Clipping Algorithm

Note that even if the algorithm does not stop, the result for B converges to some box, which we denote by B^*. The reason for that is that the clipped boxes form a series of boxes $B_1, B_2, ..$ for which holds $B_{i+1} \subseteq B_i$. The existence of the limit for this series can be easily verified as a generalization of the convergence of series of intervals.

Part 1 of Theorem 2: *The clipping algorithm never yields a false negative.*

Proof The description of the algorithm in Appendix C (with the safe dot product algorithm) includes the proof. ∎

Part 2 of Theorem 2: *For $n = 2$, if $P \neq \emptyset$ the algorithm reaches the correct result after one iteration (no false positives).*

Proof Let B_P be the tight bounding box of P (although we do not have it as a part of the input). If we have $B \cap B_P \neq \emptyset$ then the simple test does not yield a false positive (note that the simple test does not have to use B_P in this case). Therefore, from part 1 we have that the clipping algorithm does not yield a false positive. Suppose $B \cap B_P = \emptyset$. WLOG suppose $h_2 < L_2$. Let $p = (x, L_2)$ be a vertex of P. We know such a vertex exists because L_2 is the low bound of B_P which is a tight bounding box of P. The possibilities for constraints that form this vertex are grouped into two cases:

Case 1: There is a constraint of the form $X_2 \geq L_2$ in P. B is outside that constraint so both the simple test and the clipping algorithm would yield a "No" output and hence no false positive.

Case 2: There are two constraints in P of the form $a_1X_1 + b_1X_2 \geq a_1x + b_1L_2$ and $a_2X_1 + b_2X_2 \geq a_2x + b_2L_2$ where $a_1 > 0$, $b_1 \geq 0$, $a_2 < 0$ and $b_2 \geq 0$. When we clip B using the first of those constraints we get every point $(x', y') \in B$ that satisfies $x' > x$ (and obviously $y' < L_2$) holds $a_1x' + b_1y' < a_1x + b1L_2$ so this point is outside the first constraint. Therefore B is clipped such that its upper bound for the X_1 axis is lower than x. Similarly for the second constraint B is clipped such that the lower bound of the X_1 axis is higher than x. Therefore we have that these two constraints clip B into $\emptyset$ and the algorithm returns a "No" output. This result holds for any order that we apply the constraints so in all the algorithm will return "No" and hence does not have a false positive.

This completes the proof. ∎

Part 3 of Theorem 2: *For $n = 2$, if no bound is placed on I, the algorithm always halts with the correct answer.*

Proof Since the algorithm is conservative it gives the correct answer if $B \cap P \neq \emptyset$. Suppose we have that $B \cap P = \emptyset$. A point (x_1, x_2) is outside a constraint $a_1X_1 + a_2X_2 \geq c$ if $a_1x_1 + a_2x_2 < c$. The distance of that point from the half-plane defined by the constraint is $c - a_1x_1 - a_2x_2$. Every point $p \in B$ is outside P so for every such point exists a constraint s.t. the point is outside the constraint. Define $d(p)$ to be the distance of p from the farthest half-plane that defines P. Define $\epsilon = \min\{d(p)|p \in B\}$. Since $B \cap P = \emptyset$ we have that for every $p \in B$ holds $d(p) > 0$. Since B is a closed region exists $p \in B$ s.t. $d(p) = \epsilon$. Therefore $\epsilon > 0$. Let $a_1X_1 + a_2X_2 \geq c$ be a constraint in P s.t. the center of B (i.e. $((l_1 + h_1)/2, (l_2 + h_2)/2)$) has distance δ from the constraint and $\delta \geq \epsilon$. We can reflect the B and P around any axis and get the same geometric problem so WLOG we have $a_1 \geq 0$ and $a_2 \geq 0$. We have that the distance of the vertex (l_1, l_2) of B from the constraint is at least δ. Define δ' to be the distance of (l_1, h_2) from the constraint so $a_1l_1 + a_2h_2 \geq c + \delta'$. (If the point is in the constraint then $\delta' = 0$, hence the inequality.) Define δ'' to be the distance of (h_1, l_2) from the constraint so $a_1h_1 + a_2l_2 \geq c + \delta''$. We have that $\delta' + \delta'' \leq a_1(l_1 + l_2) + a_2(l_1 + l_2) - 2c$. The fact that the center of B has distance δ from the constraint yields $a_1(l_1 + l_2) + a_2(l_1 + l_2) = 2c + 2\delta$ so we have $\delta' + \delta'' \leq 2\delta$. Therefore either $\delta' \geq \delta$ or $\delta'' \geq \delta$. If $\delta' \geq \delta$ then B is clipped by increasing l_1 by at least δ. If $\delta'' \geq \delta$ then B is clipped by increasing l_2 by at least δ. Therefore in any case at least one of the sides of B is shortened by at least ϵ. The result of the first iteration of clipping is B'. Since $B' \subset B$ we have that $\min\{d(p)|p \in B'\} \geq \min\{d(p)|p \in B\} = \epsilon$ so in each iteration we clip at least ϵ from one of the sides or we get $\emptyset$ as a result. Since ϵ remains a constant through all the process we have that after a finite number of clips we end with $\emptyset$ and do not return a false positive. ∎

Part 4 of Theorem 2: *For any $n \geq 2$ and any $K > 0$, we can always find some input on which the algorithm stops after exactly $K + 1$ iterations when no bound is placed on the number of iterations I.*

Proof The example in Appendix C is in two dimensions and the number of iterations depends of a single input parameter ϵ. We an create such an example in any number of dimensions by using these constraints and ignoring the rest of the dimensions. For every K exists some ϵ such that the number of iterations is some J where $J > K$. Let B' be the result after $J - K - 1$ iterations. The same problem using B' instead of B as input will stop after exactly $K+1$ iterations. ■

Part 5 of Theorem 2: *For $n > 2$, there are cases when the algorithm stops with a false positive even when I is unbound.*

Proof Consider the case where $B = [-1, 1]^3$ and we have two constraint: $X_1 + X_2 + X_3 \geq 1$ and $X_1 + X_2 + X_3 \leq -1$. Obviously we have $P = \emptyset$ so $B \cap P = \emptyset$. On the other hand the clipping algorithm does not perform a single clip in any dimension so after one iteration it stops with a false positive. ■

Part 6 of Theorem 2: *Even when $I = 1$, this algorithm is strictly more accurate than the simple test.*

Proof Note that the first three steps in the inner loop of the clipping algorithm (as described in Section 2.4 are exactly the simple test. Therefore the clipping algorithm includes the simple test. ■

Part 7 of Theorem 2: *Suppose $P_1, .., P_m$ are linear constraints and $Q_1, .., Q_m$ are the same constraints in a different order (i.e., $\{P_1, .., P_m\} = \{Q_1, .., Q_m\}$). Suppose B is some hyper-box. Suppose B' is the result of the clipping algorithm using $P_1, .., P_m$ on B with I iterations and B'' is the result of the clipping algorithm using $Q_1, .., Q_m$ on B with mI iterations. Then $B'' \subseteq B'$.*

Proof First we make the observation:
If $B_1 \subseteq B_2$ are hyper-boxes, P is a linear constraint, B'_1 is the result of clipping B_1 with P, and B'_2 is the result of clipping B_2 with P, then $B'_1 \subseteq B'_2$.
Suppose H is the half-space that satisfies P. B'_1 is the MBH of $B_1 \cap H$ and B'_2 is the MBH of $B_2 \cap H$. Since $B_1 \subseteq B_2$ then $B_1 \cap H \subseteq B_2 \cap H$. Therefore their MBH's follow the same inclusion and $B'_1 \subseteq B'_2$.

The sequence of constraints used by the clipping algorithm with $P_1, .., P_m$ and I iterations is:

$$\mathcal{S}_1 = [P_1, .., P_m, \overbrace{\cdots\cdots}^{I \text{ times}}, P_1, .., P_m]$$

The sequence of constraints used by the clipping algorithm with $Q_1, .., Q_m$ and mI iterations is:

$$\mathcal{S}_2 = [Q_1, .., Q_m, \overbrace{\cdots\cdots}^{mI \text{ times}}, Q_1, .., Q_m]$$

B' is the result of clipping B with the constraints in sequence $\mathcal{S}_1$. B'' is the result of clipping B with the constraints in sequence $\mathcal{S}_2$. Clearly $\mathcal{S}_1$ is a subsequence of $\mathcal{S}_2$. Therefore, using the observation above, we get $B'' \subseteq B'$. ■

E Properties of the Addition Algorithms

Lemma 1: *Suppose S is an input satisfying the conditions of case 1. Suppose $\varepsilon = \sum S - sum_1(S)$ and m is the size of mantissa used for sum_1. Then, $exp(\varepsilon) \leq exp(sum_1(S)) - m$. (We assume the exponent of 0 is $-\infty$.)*

Proof We use induction on $|S|$. Suppose the theorem is valid for all sets of size smaller than N and we prove for sets of size N. Step 1 of the algorithm is trivial. In steps 2,3 and 4 we add some numbers from S. All those numbers have exponents between e and $e - k$. There are at most 2^k numbers since $|S| \leq 2^k$. This means that the exponent of the result is at most $e + k$. Since all those numbers have $2k$ bits padding we have that accumulating all those numbers into x using the $\oplus$ operator is done with no rounding error. Furthermore, all the accumulated numbers had the same sign so the exponent of the result is at least the exponent of the first number to be accumulated which is e. The calculation of x' in step 5 is the sum of at most 2^k numbers. All those numbers have exponents smaller than $e - k$ so the result exponent is smaller than e. Define $\varepsilon_1 = \sum S - x - x'$. This is the error we have so far in the calculation. Using induction hypothesis on the calculation of x' we see that $exp(\varepsilon_1) \leq exp(x') - m < e - m$. Define ε_2 as the error in calculating $x \oplus x'$. Define $e' = exp(x \oplus x')$. We have that $|\varepsilon_2| \leq 2^{e'-m}$ and $e' \geq e$. Therefore, $exp(\varepsilon_2 + \varepsilon_1) \leq exp(\varepsilon_2) \leq e' - m$. This is what we needed to prove. ■

Lemma 2: *Suppose S_1, S_2 is an input satisfying the conditions of case 2. Suppose $\varepsilon = \sum(S_1 \cup S_2) - sum_2(S_1, S_2)$ and m as before. Then, $exp(\varepsilon) \leq exp(sum_2(S_1, S_2)) - m$.*

Proof Consider what happens in sum_2. Steps 1,2,3 and 4 duplicate what happens in the first 5 steps of the algorithm sum_1 for set S_1. After step 4 (of sum_2) we have x and x' exactly the same after step 5 of $sum_1(S_1)$. Therefore, x has no error in it, and $exp(x) > exp(x')$. Define the error incurred when calculating x' as ε_x. We have $exp(\varepsilon_x) \leq exp(x') - m \leq exp(x) - m - 1$. y is calculated using sum_1. Define the error incurred in calculating y as ε_y. We have that $exp(\varepsilon_y) \leq exp(y) - m$ (from the correctness of sum_1). Since there are at most 2^k numbers in $\mathcal{S}_2$ and $exp(top(\mathcal{S}_1)) > exp(top(\mathcal{S}_2)) - k - 1$ we have that $exp(x) > exp(y) - 1$. Therefore $exp(\varepsilon_y) \leq exp(x) - m - 2$. Define $z = x' \oplus y$ and $r = x \oplus z$. The result of the algorithm is r (see step 6). Define the rounding error incurred in the addition that results in z as ε_1. We have that $exp(x) > exp(z)$, since x', y have exponents lower than $exp(x)$ and they have different signs, so their addition can't result in a higher exponent. By definition of $\oplus$ we have $|\varepsilon_1| \leq 2^{exp(z)-m}$ so $|\varepsilon_1| \leq 2^{exp(x)-m-1}$. Define the rounding error incurred in the addition that results in r as ε_2. By definition of $\oplus$ we have $|\varepsilon_2| \leq 2^{exp(r)-m}$. Define the total error in the algorithm as ε. We have that $|\varepsilon| \leq |\varepsilon_x| + |\varepsilon_y| + |\varepsilon_1| + |\varepsilon_2|$. There are two cases:

Suppose $exp(r) \geq exp(x)$. Therefore $exp(\varepsilon_y) \leq exp(r) - m - 1$ and $exp(\varepsilon_x) \leq exp(r) - m$ and $|varepsilon_1| \leq 2^{exp(r)-m-1}$. Therefore $|\varepsilon| < 2 \cdot 2^{exp(r)-m} + 2 \cdot 2^{exp(r)-m-1} < 2^{exp(r)-m+2}$. So we have $exp(\varepsilon) \leq exp(r) - m + 1$, and this is the required result.

Suppose $exp(r) = exp(x) - 1$. Therefore the sign of z si the same as the sign of y. This means that $|x'| \leq |y|$ and $|z| \leq |y|$. Therefore $exp(\varepsilon_x) \leq \varepsilon_y$ and $|\varepsilon_1| \leq 2^{exp(y)-m} \leq 2^{exp(x)-m-2}$. In all we have $|\varepsilon| < 3 \cdot 2^{exp(r)-m} + 2^{exp(r)-m-1} < 2^{exp(r)-m+2}$. This is the same result as in the previous case.

■

Lemma 3:

1. *There is no rounding error in steps 4-9. (i.e., the additions done in steps 5,6 are without rounding errors.)*

2. *After going through steps 4-9 one of the following happens:*
   (a) *$|S_1 \cup S_2|$ is reduced.*
   (b) *The condition for using sum_1 is satisfied.*
   (c) *The condition for using sum_2 is satisfied.*

Proof

Part 1: Define $e = exp(top(S_1))$ as used in steps 4,5. Note that S_1 doesn't change through those steps so we can use e as a constant. All the numbers added in x in step 5 have exponent between $e - 2k$ to e. At all times (including after step 5) we have $exp(x) \leq e$. We also have $2k$ bits padding for all the numbers added. Therefore, we do not have any rounding errors in the additions of step 5. In step 6 we add two numbers with opposite signs. The number from S_1 has exponent e. $e - 2k \leq exp(x) \leq e$. Since all the numbers accumulated in x had $2k$ bits padding and exponent at least $e - 2k$ this addition is without error as well. (Note that if the signs are the same this does not necessarily hold.) Therefore, in steps 4-6 there is no rounding error.

Part 2: There are three possible cases:

1. The loop in step 5 was not empty. In this case we used in steps 5,6 at least three numbers from S_1, S_2. In steps 7-9 we insert back at most two numbers. Therefore, $|S_1 \cup S_2|$ is reduced.
2. There is no split in steps 7-9. This means that in steps 7-9 we insert one number into S_1 or S_2. We always use at least two numbers in steps 4-6 so $|S_1 \cup S_2|$ is reduced.
3. The loop in step 5 was empty and there was a split in steps 7-9. There are three possibilities for the reason why the loop was empty:
   (a) When we start step 5 we have $exp(x) = exp(top(S_1))$. This means that in step 6 we add two numbers with the same exponent, a different sign and both have $2k$ bits padding. The result must have $2k$ bits padding. This means that there is no split in steps 7-9 so this case can't happen.
   (b) When we start step 5 we have that S_2 is empty. We also have that $exp(x) < exp(top(S_1))$ so the result of step 6 is inserted into S_1 in steps 7-9. This means that after step 9 S_2 is empty. Therefore, the condition for sum_1 is satisfied.
   (c) When we reach step 5 we have $exp(top(S_1)) - exp(top(S_2)) > 2k$. Define $e = exp(top(S_1))$ during step 5. After step 6 the exponent of the result is e' and the result will be put in S_1. Therefore, the maximal exponent in S_1 after steps 4-9 is at least e'. The maximal exponent in S_2 after steps 4-9 is at most $e'' = e - 2k - 1$. If $e' \geq e - 1$ it means that $e' - e'' \geq e - e'' - 1 \geq 2k > k + 1$. [3] Thus, the condition for sum_2 is satisfied. If $e' < e - 1$ it means that $exp(x) \geq e - 1$ (before the addition in step 6). This means that we add two numbers with the same padding and both numbers have exponents at least as high as the result. This means that the result has at least the same amount of padding. Thus, *split* doesn't occur in steps 7-9. So this case can't happen. In all we see that we satisfy the condition for sum_2.

■

[3] k is at least 2, otherwise $|\mathcal{S}| \leq 1$ and we don't need an addition algorithm.