Holes in Joins

Jarek Gryz Dongming Liang

Department of Computer Science
York University
Toronto, Canada

August 31, 2006

Abstract

A join of two relations in real databases is usually much smaller than their cartesian product. This means that most of the combinations of tuples in the crossproduct of the respective relations do not appear together in the join result. We characterize these combinations as ranges of attributes that do not appear together. We sketch an algorithm for finding such combinations and present experimental results from real data sets. We then explore two potential applications of this knowledge in query processing. In the first application, we model empty joins as materialized views, we show how they can be used for query optimization. In the second application, we propose a strategy that uses information about empty joins for an improved join selectivity estimation.

1 Introduction

A join of relations in real databases is usually much smaller than their Cartesian product. For example, the OLAP Benchmark [11] with a star schema of six dimension tables with, respectively, 12, 15, 16, 86, 1000, and 10,000 tuples, has a fact table of the size of 2.4 millions tuples. The size of the fact table is thus 0.00009% of the size of the Cartesian product of the dimension tables.

This rather trivial observation about the relative size of the join and the respective Cartesian product, gives rise to the following questions: Can the non-joining portions of the tables be characterized in an interesting way? If so, can this knowledge be useful in query processing? Consider the following example.

Example 1 *Consider* **Lineitem** *and* **Order** *tables in TPC-H [40]. The* **o_order-date** *attribute in the* **Order** *table stores information about the time an item was ordered, the* **l_shipdate** *attribute in the* **Lineitem** *table stores information about the time an item was shipped. The two attributes are correlated: an item cannot be shipped before it is ordered and it is likely to be shipped within a short period of time after it is ordered. This is depicted graphically in Figure 1. Assume that an item is always shipped within a year from the time it is ordered. Thus, for a given range of* **o_orderdate***, only the tuples from that range extended by one year of* **l_shipdate** *will be in the join of* **Lineitem** *and* **Order***. None of the crossproduct between the remaining portions of the tables will appear together in the join result.*

Call any query that involves a join and that evaluates to the empty table an *empty join*. Knowledge of empty joins may be valuable in and of itself as it may reveal unknown correlations between data values which can be exploited in applications. For example, if a DBA determines that a certain empty join is a time

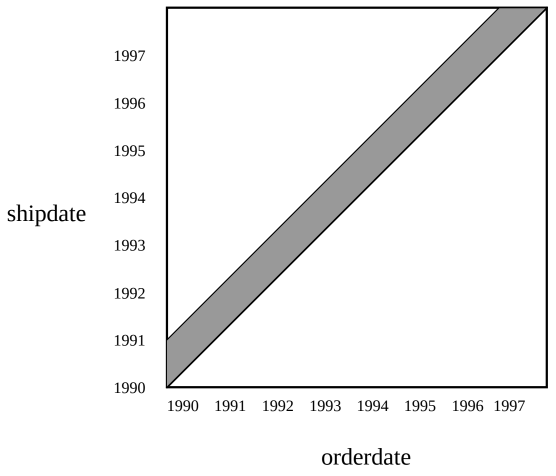

Figure 1: Distribution of tuples with respect to the values of **shipdate** and **orderdate**.

invariant constraint, then it may be modeled as an integrity constraint. Indeed, the fact that an item cannot be shipped before it is ordered is defined in TPC-H as a check constraint [40].

But even if the discovered empty joins are not the result of a time invariant property or constraint, knowledge of these regions may be exploited in query processing.

Example 2 *Consider the following query Q over TPC-H.*

```
select   sum(l_totalprice)
from     lineitem l, order o
where    l_orderkey = o_orderkey
         AND o_orderdate BETWEEN '1995.01.01' AND '1996.01.01'
```

Given the correlation of Figure 1, the original query can be simply rewritten as Q':

```
select   sum(l_totalprice)
from     lineitem l, order o
where    l_orderkey = o_orderkey
         AND o_orderdate BETWEEN '1995.01.01' AND '1996.01.01'
         AND l_shipdate BETWEEN '1995.01.01' AND '1997.01.01'
```

By reducing the range of one or more of the attributes or by adding a range predicate (hence reducing an attribute's range), we reduce the number of tuples that participate in the join execution thus providing optimization. In the extreme case, when the predicates in the query fall within *the ranges of an empty region, the query would not have to be evaluated at all, since the result is necessarily empty.*

The information about the correlation between the two attributes can also be used for a different purpose. Consider again the original query Q. Since there is no predicate (except for the join) placed on the lineitem table, it seems that any tuple from that table can potentially appear in the answer. With this assumption, a database optimizer would vastly overestimate the cardinality of the join result. Given the correlation *of Example 1, however, we can infer that only the tuples satisfying the condition* l_shipdate BETWEEN '1995.01.01' AND '1997.01.01' *can appear in the join. This knowledge can be used to provide a more exact estimate of the join selectivity.*

Note that the rewrite from Q to Q' requires that the correlation between the attributes be exact; for the purpose of cardinality estimates, an approximate correlation is sufficient, since statistical information need not be exact.

An empty join can be characterized in different ways. The most straightforward way is to describe it negatively by defining a correlation between data points that *do* join. Thus, for the two attributes from Example 1 we can specify their relationship as a linear correlation: *l_shipdate = o_orderdate + [0, 1] year*, where $[0, 1]\ year$ is the correlation error. We explored this idea in [16] and showed how such correlations can be used in query optimization. We also learned, however, that such correlations are rare in the real data that we explored. In this paper, we are proposing an alternative, but complementary approach to characterizing empty joins as ranges of attributes that *do not* appear together in the join. For example, there are no tuples with *l_orderdate* $>$ '1995.01.01' and *l_shipdate* $<$ '1995.01.01' in the join of **Lineitem** and **Order**. In other words, the join of **Lineitem** and **Order** with thus specified ranges of *l_orderdate* and *l_shipdate* is empty. To maximize the use of empty joins knowledge, our goal in this work is not only to find empty joins in the data, but to characterize fully that empty space. Specifically, we discover the set of all maximal empty joins in a two dimensional data set. Maximal empty joins represent the ranges of the two attributes for which the join is empty and such that they cannot be extended without making the join non-empty.

We suggest that empty joins can be thought of as a novel characteristic of data skew. By characterizing ranges of attributes that do not appear together, we provide another description of data distribution in a universal relation. In this paper, we show how the knowledge of empty joins can be used in query processing. The techniques we present here are straightforward generalizations of Example 2. The first technique is a rewrite-based query optimization. The second one offers a new method for improved join selectivity estimates. Although both techniques utilize information about empty joins, they do not depend upon each other and can be used separately. We show these techniques to be useful in practice by experimental verification of the following claims.

- First, real data sets contain a large number of empty joins, some of which are themselves very large. This is important as the value of our techniques increases as the data is more skewed in that sense.
- Second, the types of rewrites we propose in the first technique indeed provide powerful optimization of query execution. We present experiments showing how the quality of optimization depends on the types and number of empty joins used in a rewrite.
- Third, the join cardinality estimates provided by the second technique are almost uniformly more accuarate than estimates based on an assumption of uniform data distribution or histograms.
- Last but not least, we develop these techniques with a possible commercial implementation in mind. We show how the existing tools in DB2 can be used to implement both techniques. Our solution therefore has the highly desirable property that it provides new optimizations method without requiring any change to the underlying query optimization and processing engine.

The paper is organized as follows. In Section 2, we introduce formally the notion of an empty join and briefly describe an algorithm for their discovery. Related work is described in Section 3. In Section 4 we present the results of experiments performed on real data, showing the nature and quantity of empty joins that can occur in large, real databases. In Section 5, we describe a technique illustrating how knowledge of empty joins can be used in join size estimation. In Section 6, we evaluate the quality of join cardinality estimates based on the knowledge of empty joins. Conclusions and future work are presented in Section 7.

2 Discovery of Empty Joins

In this section we introduce a formal representation of empty joins and present a sketch of an algorithm for finding all maximal empty joins within a two dimensional data set (two-way join).[1]

Consider a join of two relations $R \bowtie S$. Let A and B be attributes of R and S respectively over two totally ordered domains. (Note that A and B are *not* the join attributes.) We are interested in finding ranges of A and B for which the join $R \bowtie S$ is empty. Define the data set $D = \Pi_{R.A,S.B}(R \bowtie S)$. Let X and Y denote the set of distinct values for attributes A and B respectively. The set D consists of a set of tuples $\langle x_i, y_j \rangle$ over two ordered domains. We can depict the data set as an $|X| \times |Y|$ matrix M of 0's and 1's. There is a 1 in position $\langle i, j \rangle$ of the matrix if and only if $\langle x_i, y_j \rangle \in D$ where x_i is the i^{th} smallest value in X and y_j the j^{th} smallest in Y. An empty join is represented in M as a rectangle containing only 0's and no 1's. The coordinates $(x_0, x_1), (y_0, y_1)$ of the rectangle specify the endpoints of the ranges of attributes A and B for which the join is empty. Since there is a one-to-one correspondence between an empty join and a 0-rectangle in the corresponding matrix M, we will sometimes refer to empty joins as empty rectangles in the remainder of this paper.

An empty rectangle is *maximal*[2] if it cannot be extended along either the X or Y axis because there is at least one 1-entry lying on each of the borders of the rectangle.

Example 3 *Let A be an attribute of R with the domain $X = (1, 2, 3)$ and let B be an attribute with domain $Y = (6, 7, 8)$. Assume that $\pi_{R.A,S.B}(R \bowtie S) = \{(3, 6), (1, 7), (3, 8)\}$. The matrix M for the data set is shown in Figure 2a. Figure 2b shows all maximal empty rectangles.*

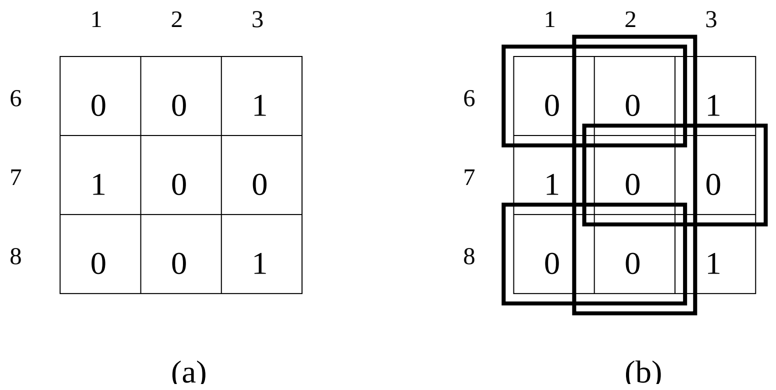

Figure 2: The matrix and some of the empty rectangles (marked with thick lines) for Example 3

Knowledge of large empty rectangles in a data set can help query optimization because such regions do not need to be considered during query processing. If a query can be optimized with respect to some empty join J, then it can be optimized at least as well with respect to a larger empty join J' that contains

[1]We make the restriction to two-way joins only for simplicity. Indeed, empty joins can be discovered for any pair of attributes from a multi-way join.

[2]It is important here not to confuse maximal with maximum (largest).

J (but not vice versa). In addition, having many empty rectangles, even if they overlap, enhances the query optimization potential of the discovered regions. For these reasons, we consider the problem of finding all maximal empty rectangles. (In practice we keep only those that are sufficiently large.)

Although it appears that there may be a huge number of overlapping maximal rectangles, [29] prove that the number is at most $\mathcal{O}(|D|^2)$, and that for a random placement of the 1-entries, the expected value is $\mathcal{O}|D| \log |D|$. We proved in [13] that the number is at most $\mathcal{O}(|X||Y|)$.)

A related problem attempts to find the minimum number of rectangles (either overlapping or not) that covers all the 0's in the matrix. (It is a special case of the problem known as Rectilinear Picture Compression [14].) This problem is NP-complete, and hence is impractical for use in large data sets. Besides, as we shall show in Section 5 that for the purpose of query optimization, it is more important to have large rectangles than to have the minimum number of rectangles.

The algorithm for finding all maximal empty rectangles in a given data set is scalable to large data sets because it uses relatively little memory and keeps disk access to a minimum. The input consists of a two dimensional data set D of tuples $\langle x_i, y_j \rangle$ stored on disk and sorted with respect to the Y domain. The algorithm requires only a single scan over this data. The output consists of the coordinates of the empty rectangles and can be written to disk, as generated. The memory requirements are $\Theta(|X|)$, which is an order of magnitude smaller than the size $\mathcal{O}(|X||Y|)$ of both the input and the output. (We assume without loss of generality that $|X| \leq |Y|$.) The time complexity of the algorithm, $\mathcal{O}(|X||Y|)$, is linear in the size of the underlying matrix.

The matrix representation M of the data set D is never actually constructed. For simplicity, however, we describe the algorithm completely in terms of M. (The reader is referred to [13] for the details of the algorithm). We shall ensure that only one pass is made through the data set D.

The main strategy of the algorithm is to consider each 0-element $\langle x, y \rangle$ of M one at a time, row by row. Although the 0-elements are not explicitly stored, this is simulated as follows. We assume that the set X of distinct values in the (smaller) dimension is small enough to store in memory. The data set D is first sorted with respect to the Y domain. Tuples from D are read sequentially off the disk in this sorted order. When the next tuple $\langle x_i, y_j \rangle \in D$ is read from disk, we are able to deduce the block of 0-elements in the row before this 1-element.

When considering the 0-element $\langle x, y \rangle$, the algorithm needs to look ahead by querying the matrix elements $\langle x+1, y \rangle$ and $\langle x, y+1 \rangle$. This is handled by having the single pass through the data set actually occur one row in advance. Similarly, when considering the 0-element $\langle x, y \rangle$, the algorithm looks back and queries information about the parts of the matrix already read. To avoid re-reading the data set, all such information is retained in memory.

The algorithm sketched above has a straightforward generalization to higher dimensions (also described in detail in [13]). However, in all experiments described in the remainder of this paper, we only consider two dimensional datasets. This is a consequence of the fact that the number of maximal hyper-rectangles in a d-dimensional matrix is $\Theta(n^{2d-2})$ (where $n = |X| = |Y|$). The number of such maximal hyper-rectangles (which represent multi-way empty joins) and hence the complexity of an algorithm to produce them increases exponentially with d. For $d = 2$ dimensions, this is $\Theta(n^2)$, which is linear in the size $\Theta(n^2)$ of the input matrix. For $d = 3$ dimensions, it is already $\Theta(n^4)$, which is not likely practical in general for large data sets. A heuristic is required to discover hyper-rectangles. We do not address this issue here.

3 Related Work

We are not aware of any work on discovery or application of empty joins.

Extracting semantic information from database schemas and contents, often called *rule discovery*, has been studied over the last several years. Rules can be inferred from integrity constraints [5, 4, 41] or can be discovered from database content using machine learning or data mining approaches [8, 10, 19, 34, 36, 41]. It has also been suggested that such rules be used for query optimization [20, 34, 36, 41]. None of this work, however, addressed the specific problem we solve here.

Another area of research related to our work is answering queries using views. Since we model empty joins as a special case of materialized views (that are also empty), essentially all techniques developed for maintaining, and using materialized views for query answering apply here as well [17, 25, 7, 38].

Also, since empty regions describe semantic regularities in data, they are similar to integrity constraints [15]. They describe what is true in a database in its current state, as do integrity constraints, but can be invalidated by updates, unlike integrity constraints. Using empty joins for query optimization is thus similar to semantic query optimization [6, 18, 22, 23, 35, 9], which uses integrity constraints for that purpose.

Query optimizer makes heavy use of the statistical information in cardinality estimation. There are two ways to store such information in the database: parametric and non-parametric [26] . In the parametric approach, the actual value distribution is approximated by a parameterized mathematical distribution. This technique requires little overhead, but it is typically inaccurate because real data does not usually follow any known distribution. Non-parametric approach is often histogram-based [32, 31, 28, 21, 27]. While a histogram is adequate for one attribute on a base table, [24] shows that a histogram is not practically efficient for multiple columns because of high storage overhead and high error rates. Current commercial database systems usually maintain histograms only for individual columns on base tables.

A query joining two or more tables with multiple columns referenced makes the situation even more complex. To estimate the size of such queries, the optimizers need to assume independence between attributes and predicates, and errors in the estimates may increase exponentially with the number of joins [15]. The problem is typically caused by propagating statistical information through the query plan. As a result, the optimizers often return low-quality execution plans for complex join queries.

We are thus motivated to propose building new statistics over non base-relations for better estimates of join cardinality. To the best of our knowledge, there is no in depth effort so far to address this type of problem. [1] presents join synopses based on sampling for approximating query processing, and the technique is restricted to be foreign-key joins. In contrast, we focus on estimating query cardinality. We are not histogram-based or sample-based, and we place no restriction on the type of joins.

4 Characteristics of Empty Joins

We would expect real data sets to exhibit different characteristics than synthetic data sets such as the TPC-H benchmark. Hence, to characterize empty joins we used two real databases: the first, an insurance database; and the second, a department of motor vehicles database. We ran the empty joins mining algorithm on 12 pairs of attributes. The pairs of attributes came from the workload queries provided with the databases. These were the attributes frequently referenced together in the queries (one from one table, and the other from a second table, and the tables are joined). For conciseness, we only present the results of four representative tests here.[3] For all reported tests the mining algorithm ran in less than 2 minutes (on a single-user 67MHz IBM RISC System/6000 machine with 512 MB RAM).

[3]They are representative in the sense that they cover the spectrum of results in terms of the number and sizes of the discovered empty joins.

Test	n	m	T	R	S (sizes of the 5 largest empty joins				
					1	2	3	4	5
1	525	3683	269	8	74	73	69	7	7
2	6	37716	39572	29323	68	58	40	37	28
3	3	1503	3061	650	97	94	80	12.2	0.04
4	525	423	42854	13850	91.6	91.6	91.3	91.3	83.1

Table 1: Data characteristics

Table 1 contains the mining results: the number of distinct values, n and m, of each of the attributes, the number of tuples in the dataset T (this is the number of 1-entries in the matrix representation of the dataset), the total number of empty joins R, and the sizes of the five largest empty joins. The size metric S defines the size of an empty join as the area it covers with respect to the domains of values of the two attributes. It is defined formally in the following way.

Let E be an empty join with the coordinates $(x_0, y_0), (x_1, y_1)$ over attributes A and B with sets of distinct values X and Y respectively in tables T_1 and T_2 respectively. The relative size of the join with respect to the covered area, S, is defined as:

$$S(E) = \frac{(x_1 - x_0) * (y_1 - y_0)}{[max(X) - min(X)] * [max(Y) - min(Y)]} \tag{1}$$

We make the following observations:

1. The number of empty joins discovered in the tested data sets is very large. In some cases (see Test 3) it is on the order of magnitude of the theoretical limit of the possible number of empty joins [13].

2. In virtually all tests, extremely large empty joins S were discovered. Usually, however, only a few are very large and the sizes drop dramatically to a fraction of a percentage point (see Figure 3) for the others.

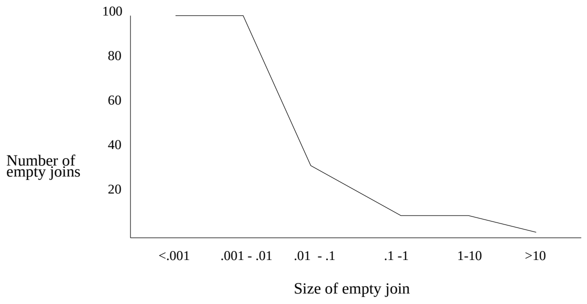

Figure 3: Distribution (with respect to S) of empty joins in Test 1.

3. The empty joins overlap substantially. The five largest empty joins from Test 1 overlap with, respectively, 7, 11, 16, 7, and 8 other empty joins discovered in that data set. These overlaps are a consequence of our decision to find *all* maximal empty joins. They also cover a large area of the join matrix; that is, the combination of values from the domains of the two attributes. The white area of

Figure 4 indicates the combinations of values of attributes A and B for which tuples exist in the join result.

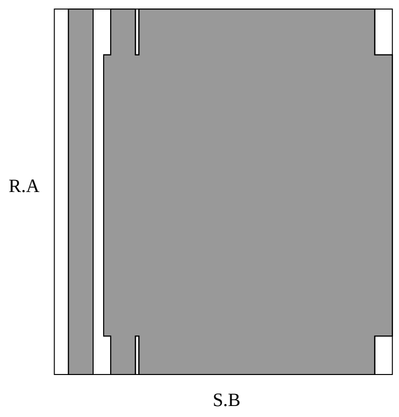

Figure 4: The area (in dark) covered by the five largest empty joins in Test 1 indicates no tuples in the join for the respective ranges of attributes A and B.

4. The experiments reveal a new type of data skew: non-uniformity of data distribution within a join result. This non-uniformity is extreme in the sense that no tuples exist within wide ranges of specified attributes.

5 Using Empty Joins in Query Optimization

5.1 Query Rewriting

We now turn to the question of how to effectively use the knowledge about empty joins in query optimization. Our approach is to model the empty joins as materialized views. The only extra storage required is the storage required for the view definition since the actual materialized view will be empty.

Assume that the following query represents an empty join.

select $*$
from R, S
where $R.J = S.J$
and X between x_0 **and** x_1
and Y between y_0 **and** y_1

We can represent this query as a materialized view V and use it to rewrite future queries to improve their performance. Indeed, we can use existing results on determining whether a view can be used to answer a query and on rewriting queries using such views [38]. Rather than describing the algorithm, we present an example of how it would be applied here.

For example, if $x_0 \leq x_i, x_j \leq x_1$ and $y_0 \leq y_i \leq y_1 \leq y_j$, then the following rewrite of query Q is possible.

select $*$
from R, S

select $*$
from R, S

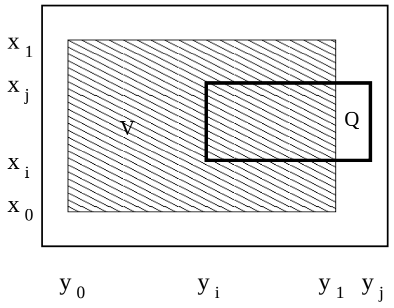

Figure 5: Query Q (marked with thick lines) overlaps the empty View V.

where $R.J = S.J$	$\Rightarrow$	*where* $R.J = S.J$
and X between x_i **and** x_j		**and** X between x_i **and** x_j
and Y between y_i **and** y_j		**and** Y between y_1 **and** y_j

This rewrite can be graphically represented, as shown in Figure 5, as a reduction of the area of the rectangle representing query Q with respect to the attributes X and Y, given the overlap with the empty join V.

Effectively, we are using the empty joins to reduce the ranges of the attributes in the query predicates. This, in turn, reduces the size(s) of the tables participating in the join, thus reducing the cost of computing the join.

5.2 Choosing Among Possible Rewrites

There are several ways such rewrites of the ranges can be done depending on the types of overlap between the ranges of the attributes in the query and the empty joins available. Previous work on rewriting queries using views can be used to decide when a view, in this case an empty view, can be used to rewrite the query [25, 38]. However, this work does not give us a way of enumerating and prioritizing the possible alternative rewrites for the inequality predicates used in our queries and views.

As shown in Figure 5, a pair of range predicates in a query can be represented as a rectangle in a two dimensional matrix. Since the goal of the rewrite is to "remove" (that is, not to reference) the combination of ranges covered by an empty join, we need to represent the non-empty portion of the query, which we will call the *remainder query* [12]. Consider Figure 6, which illustrates five fundamentally different ways a query, represented as a rectangle with thick lines, can overlap with an empty join marked as a filled rectangle.

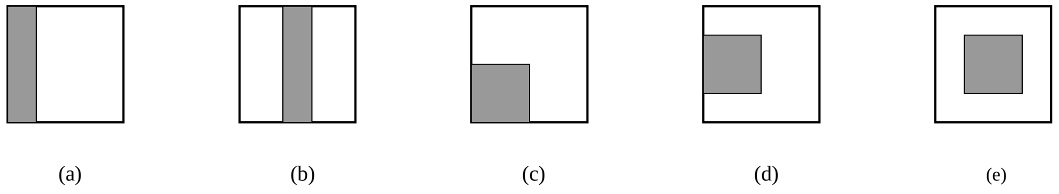

Figure 6: Overlaps of increasing complexity between the query and empty joins.

In Case (a), the remainder query can be still represented as a single rectangle. Hence, the rewritten SQL query has a single non-empty query block. In Case (b), however, the remainder query has to be represented by at least two rectangles, which implies that the rewritten SQL query will be the UNION ALL of two non-empty query blocks (or an appropriate OR condition). Cases (c), (d), and (e) illustrate even more complex scenarios where three or four rectangles are needed to describe the remainder query. Indeed, it has been shown in [7] that blind application of materialized views may result in worse plans compared to alternative plans that do not use materialized views. This is also true about empty views. Our experiments reported below suggest that using rewrites containing multiple non-empty query blocks usually degrade rather than improve query performance. The decision about which empty joins to use in a rewrite must be made within the optimizer in a cost-based way. There are cases, however, when cost-based optimization can be avoided. For example, a rewrite of type (a) in Figure 8 is guaranteed not to produce worse performance than in the original query as the only change it makes in the query is to reduce the exisiting range predicates. Our experiments also showed that a rewrite of type (b) consistently delivered optimization.

In the next subsection we investigate how the following factors affect the quality of optimization:

1. The size of the overlap between an empty join and a query.
2. The type of the overlap.
3. The number of empty joins overlapping the query used in the rewrite.

To demonstrate the usability of empty joins for query optimization under various overlap conditions, we performed several sets of experiments.

5.3 Factors Affecting Optimization

The experiments described below were run on a PC with PII-700MHz, 320M Memory under Windows 2000, DB2 UDB V7.1 for NT.

We created two tables $R(id\,int, X\ int, J\ int)$ and $S(id\,int, X\ int, J\ int)$, where J is a join column and X and Y are attributes with totally ordered domains. The range of values for both X and Y is $0 - 10,000$. R has 100k tuples, S has 10M tuples uniformly distributed over the domains of X and Y (hence $S_A = S_T$). The join method used in the queries below is sort-merge join.[4] No indexes have been created or used.

In all experiments the query had the form:

select $*$
from R, S
where $R.J = S.J$
and X between $4,000$ **and** $6,000$
and Y between $2,000$ **and** $8,000$

In the first experiment, we created empty joins with an increasing overlap with query. This was done by changing the values of one of the join attributes so that the tuples in the designed range do not join with any tuples in the other table. The empty joins had the following form:

create view empty as
select $*$

[4]Similar results were obtained for nested-loops join. No optimization can be achieved for index nested-loops, since the join is executed before the selections.

from R, S
where $R.J = S.J$
and X between $4,000$ **and** $6,000$
and Y between $2,000$ **and** par

with par set to : 2,300, 2,600, 3,250, 3,500, 4,000, 5,000, 6,000, and 7,000. The overlaps of the query and the empty join are graphically presented in Figure 7.

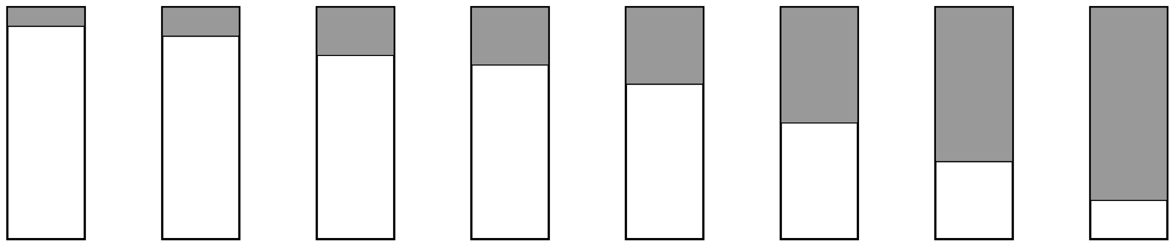

Figure 7: Increasing overlaps between a query and an empty join.

Reduction of the size of the table(%)	5	10	20	25	33	50	66	83
Reduction of execution time (%)	2.4	6.6	16	39	41	48	56	67

Table 2: Improvement in query execution time (in %) as the overlap with the empty join is increased.

As we expected, the reduction in query execution time grows monotonically with the increase of the overlap. On the other hand, the size of the overlap does not provide equivalent reduction in the query execution time. This is understandable, as the query evaluation involves not only the join execution, but also scanning of the two tables which is a constant factor for all tests. The only surprising result came from Test 4 (and later in Test 5): the reduction of the query execution time jumps above the reduction of the table's size. As it turns out, the table became sufficiently small to be sorted in memory, whereas before it required an external sort.

In the second experiment we kept the size of the overlap constant, at 25% of the size of the query, but changed the type of an overlap as shown in Figure 8.

Type of overlap	a	b	c	d	e
Reduction of execution time (%)	39	37	4.6	0	-13

Table 3: Impact of the type of the overlap used in rewrite on query performance.

As shown in Table 3, only the first two types of the overlap provide substantial performance improvement. As the number of OR conditions (or UNION's) necessary to express the remainder query increases, the performance deteriorates. For example, in Case (e), the query rewritten with ORs would have the following structure:

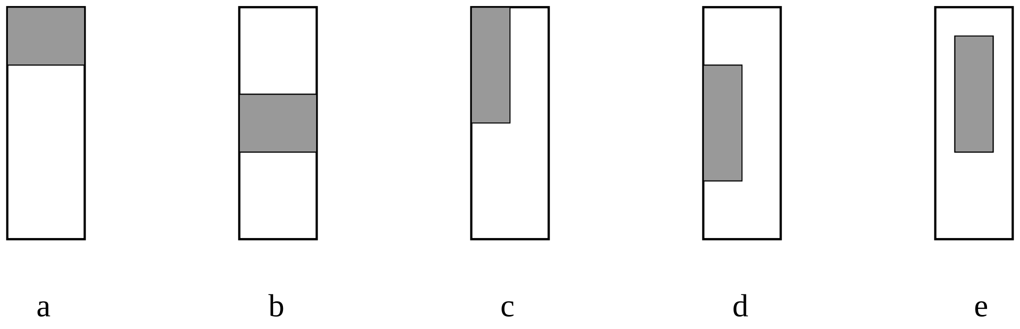

Figure 8: Overlaps of increasing complexity between the query and empty joins.

select $*$
from R, S
where $R.J = S.J$ **and**
[(X between $4,000$ **and** $6,000$
and Y between $2,000$ **and** $3,000$)
or
(X between $4,000$ **and** $6,000$
and Y between $6,000$ **and** $8,000$)
or
(X between $4,000$ **and** $4,500$
and Y between $3,000$ **and** $6,000$)
or
(X between $5,500$ **and** $6,000$
and Y between $3,000$ **and** $6,000$)]

If the remainder query were expressed using UNIONs, it would have four separate blocks. Executing these blocks requires multiple scanning of relations R and S. Indeed, in all our experiments only the rewrites using overlaps of type (a) or (b) consistently led to performance improvement.

In the third experiment we kept the size of the overlap constant at 25% and used only type (a) and (b) of the overlap from the previous experiment. This time, however, we changed the number of overlapping empty joins with the query. We varied the number of empty joins used in a rewrite from 1 to 8 decreasing their sizes accordingly (to keep the total overlap at 25%) as shown in Figure 9. The results are shown in Table 4.

Number of overlapping empty joins	1	2	4	6	8
Reduction of execution time (%)	39	38.4	38.3	35.9	34.3

Table 4: Impact of the increasing number of overlaps used in rewrite on query performance.

Interestingly, query performance degrades very slowly with the increasing number of empty joins used in a rewrite. The reason is, that despite an appearance of an increased complexity of the query after the rewrite (see the query of Test (c) below), a single scan of each table is still sufficient to evaluate the join.

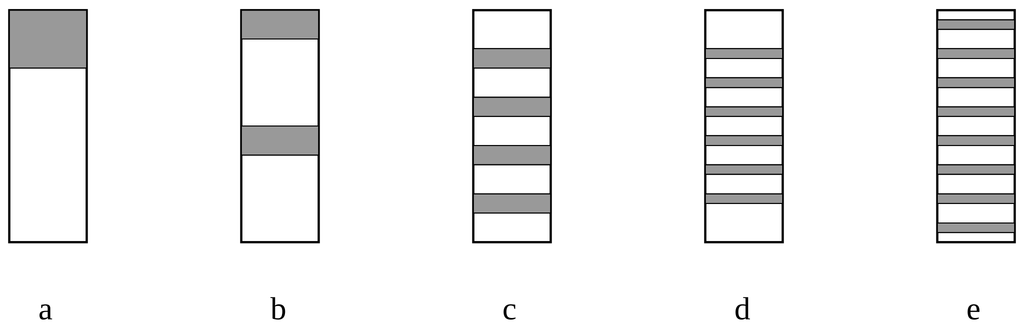

Figure 9: Overlaps with increasing number of empty joins.

select $*$
from R, S
where $R.J = S.J$ **and**
X between $4,000$ **and** $6,000$ **and**
[(Y between $2,000$ **and** $2,400$)
or
(Y between $3,000$ **and** $3,400$)
or
(Y between $4,000$ **and** $4,400$)
or
(Y between $5,000$ **and** $5,550$)]

We performed a number of experiments measuring the quality of optimization in the real workload of the insurance database using some of the empty joins described in Section 4. The results were consistent with those reported above. Ideally, we would have presented the optimization results performed with a publicly available workload, such as TPC-D, to allow for replication of the experiments. Unfortunately, the type of data distribution in TPC-D is not representative of a real data set. The data is synthetically generated, and the distribution of the attribute values tends to be uniform. We ran the mining algorithm on several pairs of attributes on TPC-D joins. In all cases the results were very different from what we discovered in real data sets. Although the number of empty joins was large, they all were very small thereby making them impractical for use by this technique. Indeed, we believe that the optimization technique presented in this paper would be most useful in datasets with large data skew where large empty joins might exist.

5.4 Quality of Optimization in Real Queries

To confirm the effectiveness of our proposed query optimization techniques we considered real data sets. To be useful, the rewrites we propose must provide significant improvements in processing queries from a real query workload. Our goal was to verify the following conjuctures:

- The discovered empty joins overlap with queries from a real workload.
- The optimization achieved through rewrites is consistent with the results on synthetic data reported in the previous section.

To demonstrate the benefits of our optimization, we considered all queries from a workload for one of the real world databases discussed in Section 4, the insurance database. A query qualified for a rewrite if it contained range predicates and involved a join. There were 5 queries (T1,...,T5) in the workload of 30 that qualified for the rewrite.

Next, we mined for empty joins for each pair of attributes relevant to these queries. We consider queries of the following form.

select	*
from	$R_1, ..., R_N$
where	JoinCond(Q) and OtherCond(Q)
	and X between x_0 **and** x_1
	and Y between y_0 **and** y_1

For each such query, we mined the data sets produced by the following query.

select	X, Y
from	$R_1, ..., R_N$
where	JoinCond(Q)

Figure 3 in Section 4 reports the size of the discovered empty joins. The empty joins found in Test 1 were used to rewrite Query T1, in Test 2 for Queries T2 and T3, in Test 3 for Query T4, and in Test 4 for Query T5. Our results show that there is substantial overlap between the discovered empty joins and the queries. We show in Figure 10 the number and the percentage of all empty joins that overlapped with each of the five queries.

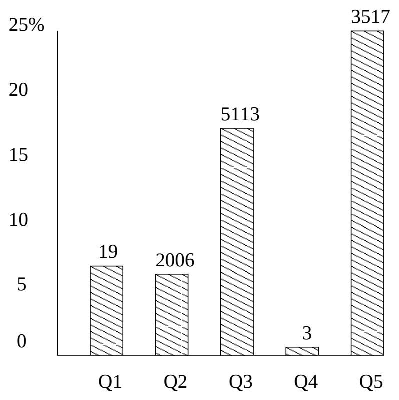

Figure 10: Percentage and the number (shown above each bar) of all empty joins overlapping with each of the queries.

Once we discovered all the empty joins, we proceeded to perform the query rewrite as described in Section 5.3. In each case, we only considered the simple rewrites (type a in Figure 8) using an empty join with the largest overlap with the query. Not surprisingly, the empty joins with the largest overlaps were also large empty joins themselves. They ranked 1, 3, 3, 17, 12 among the rectangles in the respective datasets for queries T1-T5. This is important as it indicates that only a few of the largest joins need to be kept to provide useful optimizations.

Table 5 shows the performance improvement for each of the queries rewritten in this way compared against the execution time of the original. These optimizations were consistent with our experiments on synthetic data.

Query	T1	T2	T3	T4	T5
Performance Improvement (%)	11	75	31	20	28

Table 5: Improvement in execution time (in %) of the tested queries.

5.5 Selection and Maintenance of Empty Joins

As proposed in Section 5.1, we model empty joins as materialized views. Thus, the choice of the "best" empty joins could be determined by the same considerations that are used in choosing standard materialized views for query optimization [2, 17, 3, 42]. In particular, the techniques we developed in [42] are particularly useful here as they require almost no human intervention. After an initial set of materialized views is suggested (in our case these would be a set of the n largest empty joins; n is the parameter that can be set experimentally just as it is the case with materialized views), the system automatically selects a set of views that provide the best ratio of benefit (improvement in workload query performance) versus cost (view maintenance). Both benefit and cost are computed - through estimation - by a standard database query optimizer.

Although empty joins do not take space to store (except for their descriptions), there is an associated maintenance cost. Techniques developed for the maintenance of materialized views [17] can be applied here as well. Since empty joins are a special case of materialized views, even more efficient maintenance techniques can be devised for them. For example, empty joins are immune to deletions (they may become non-maximal, but they still correctly describe empty regions). For an insertion - if it falls within a range of an empty rectangle - the rectangle can be simply divided into smaller ones (again, at the expense of losing optimality). Still, to maintain an optimal set of empty views in the face of frequent updates would be prohibitive. However, the benefit of using empty views in a warehousing environment - where updates are infrequent - could be tremendous as our experiments show. Furthermore, there is a growing trend in industry [30, 15] to store and use new forms of integrity constraints and materialized views that are not verified or updated (because no updates are expected to violate them). The need for such constraints arises from the benefits they can have in many applications, in particular, for query optimization through query rewrites. The maintenance of such constraints is essentially free.

6 Using Empty Joins to Improve Cardinality Estimates

6.1 Soft Constraints

We introduced in [15] the concept of *soft constraints*. The idea is that a soft constraint is a constraint statement which is valid with respect to the *current* state of the database. On the next update to the database—in other words, whenever the state of the database changes—the constraint may become invalid. If a transaction would invalidate an actual integrity constraint (a so-called "hard" constraint), the transaction is aborted and the consistency of the database is preserved. On the other hand, if a transaction would invalidate a soft constraint, it is not aborted for that reason. Instead, if the transaction commits, the soft constraint is "aborted" since it is no longer consistent with respect to the database. Thus consistency of the database is still maintained, but by different means.

Soft constraints are not meant to protect the integrity of the database as do integrity constraints, but like integrity constraints, they do semantically characterize the database. As integrity constraints are now used to an extent in query optimization, soft constraints can be used in the optimizer in the same way. If there are any useful characterizations of the database that are valid with respect to the current state of the database and would be useful for the optimizer with respect to the workload, but are not truly integrity constraints (that is, the database designer has no reason to specify these as rules), then these could be expressed as soft constraints. Clearly, empty joins can be modeled as soft constraints.

So far in this paper, we have only described empty joins that are valid with respect to the current state of the database in the same way standard integrity constraints are. This was necessary to guarantee soundness of query rewrites. But we can also consider empty joins - or any other soft constraints - that are "soft" in another way as well: that the soft constraint statement does not completely hold with respect to the database. Instead, some "violations" of the statement are acceptable. Thus, soft constraints can be further divided into:

- *absolute soft constraints*, which have no violations in the current state of the database (empty joins described in Section 5 are like that); and
- *statistical soft constraints*, which can have some degree of violation.

Statistical soft constraints are easier to maintain, since it does not matter when they go slightly stale. We show in this section howempty joins modelled as statistical soft constraints can be used for an improved join cardinality estimates.

6.2 The Strategy

Most commercial database systems adopt the *uniform distribution assumption* (*UDA*) [33] for estimating query result size. This assumption is often incorrect even for a single attribute and is almost never true for a joint data ditribution of two or more attributes in a relation. This non-uniformity becomes extreme when the attributes come from different relations and appear together in a join result. Histograms have been shown to be an effective tool in estimating query selectivity independently of data distribution. However, their use has been mostly limited to single attribute queries; multidimensional histograms are expensive to construct and maintain.

We are proposing a new technique (we call it *SIEQE* for *Statistics in Empty Query Expression*) for selectivity estimation of joins which provides much better prediction quality over *UDA* without the overhead associated with histograms. Our strategy is to discover and maintain several large empty joins and use information about them to improve the estimates of query selectivity. Although our techniques can be applied to distributions of several dimensions, we only consider two dimensional queries in this paper.

Let R and S be two relations and $R.A$ and $S.B$ be two attributes referenced in range predicates. The first step of the technique consists in mining the join $R \bowtie S$ for empty joins with respect to attributes A and B. Only the largest of the empty joins are maintained.[5] Next, we compute the total area covered by the empty joins and adjust the "density" of the data points in the remaining area. Let N be a number of tuples in $R \bowtie S$ and $< a_1, a_n >$ and $< b_1, b_m >$ be the ranges of A and B respectively. The density of data points (which is assumed to be uniform by *UDA*) can be defined as $D = \frac{N}{(a_n - a_1) * (b_m - b_1)}$. Let $Empty$ be the total area covered by empty joins. Then the density of data points in the remaining are should be adjusted to be $D' = \frac{N}{(a_n - a_1) * (b_m - b_1) - Empty}$. Once a query is submitted, its overlap with the empty joins is determined and the size of non-empty area calculated. The number of data points in the non-empty area is then estimated from the adjusted density D'. We illustrate the technique on following example.

[5]The decision as to *how many* of the empty joins to maintain is application dependent; just as the decision on the number of buckets in a histogram.

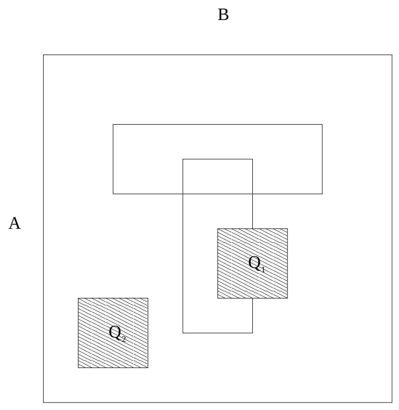

Figure 11: Empty joins and queries for Example 4.

Example 4 *Let the range of both A and B be $< 0, 100 >$ and the join contain $N = 10,000$ tuples. Assume that two empty joins have been discovered, one for $20 < A < 40$ and $20 < B < 80$, and the second one with $30 < A < 80$ and $40 < B < 60$ as shown in Figure 11. Thus, the empty joins cover 2,000 units of the entire (two-dimensional) domain. With* UDA, *the density D would be eqal to 1 tuple per square unit. With the information about the empty areas, we can infer that the density is in fact larger in the non-empty areas and equal to:*

$$D' = \frac{10,000}{(100-0)*(100-0)-2,000} = 1.25$$

Let the first query Q_1, shown in Figure 11, be:

select *
from R, S
where $R.X = S.X$
and $50 < A < 70$ *and* $50 < B < 70$

Since only half of the query is within the region containing any data points, we can estimate the number of tuples in the result to be:

$$\frac{1}{2} * (70-50) * (70-50) * 1.25 = 250$$

With UDA, *the number of tuples would have been overestimated to be 400.*

On the other hand, queries that do not overlap with empty regions would have their selectivities underestimated as the density of tuples would have been assumed to be lower. Consider query Q_2:

select *
from R, S
where $R.X = S.X$
and $10 < A < 30$ *and* $70 < B < 90$

With UDA, *the estimated number of tuples would have been 400. However, given the existence of empty joins and consequently higher density of tuples outside the empty areas, that number should be estimated to be 500.*

We note that *SIEQE* will *never* produce worse estimates than *UDA* if the data is uniformly distributed outside of empty regions. Without that assumption, however, *UDA* may happen to be more accurate for some queries.

6.3 Quality of Estimates

We performed experiments on 8 query templates[6] from the workload described in Section 4. Each query template contained a join and two range selections. For each query template, we selected randomly 100 sets of endpoints for the ranges of the two attributes and estimated the result sizes of each such query. We mined for and maintained only five largest empty joins for each pair of attributes tested.

One difficulty we faced in comparing errors produced by *SIEQE* and *UDA* was the fact that for queries which fall entirely within empty regions the error is either 0 or infinite. Thus, whenever the actual number of tuples was 0, we computed the error as if the number of tuples were equal to .01. Figure 12 shows the results. Except for one query template, *SIEQE*'s estimates are orders of magnitude better than *UDA*'s.

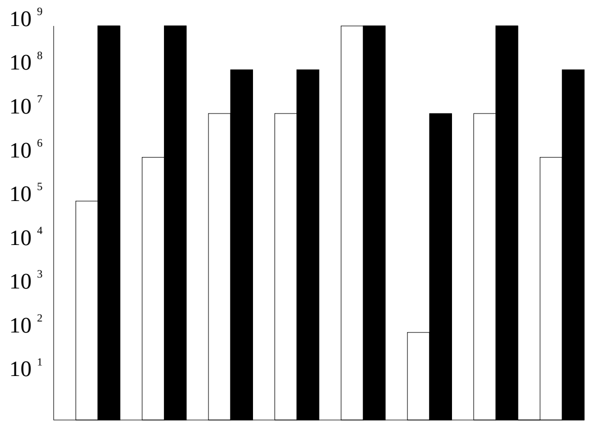

Figure 12: Average estimate error for eight query templates for *SIEQE* (white bars) and *UDA* (black bars).

The observed errors (even after empty queries are eliminated) are quite large.[7] The reason is that the data distribution outside of the empty areas is far from being uniform and there are a few queries for which the errors are enormous. However, for most of the queries the errors produced by *SIEQE* are acceptable. Figure 13 shows the proportion of queries for which the error (produced both by *SIEQE* as well as *UDA*) was less than a given limit. For over 70% of queries, *SIEQE* predicted their cardinality with less than 10% error; *UDA* achieved it only for 34% of the queries.

We also compared the prediction quality of *SIEQE* to the estimates provided by the query optimizer in a commercial system (DB2 Enterprise Server Edition, V8.1). As expected, on average, DB2 estimates were worse than either *SIEQE* or *UDA*.

We believe that our technique has important advantages over multidimensional histograms. First, it allows for an incremantal maintenance of empty joins. The only algorithm for dynamic maintenance of multidimenional histograms that we are aware of [39] does not apply to queries over joins. In the absence of incremental maintanance, multidimensional histograms have to be recomputed statically from the data. Our

[6]The workload queries did not contain actual values for the attribute ranges.

[7]We emphasize again that the number of maintained empty joins was very small (less than 5); the errors can be easily reduced by increasing that number.

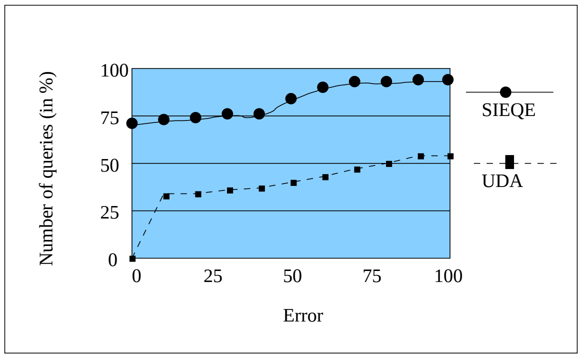

Figure 13: Number of queries for which the error was less than a given limit.

approach is also superior to histograms on that issue: constructing multidimensional histograms incurs a substantial cost in addition to computing the join (which is the input to the construction algorithm) [28, 32]. The algorithm for empty join discovery requires only a single scan of the join result.

7 Conclusions

We presented a novel approach to characterizing data that is not based on detecting and measuring similarity of values within the data, but is instead based on the discovery of empty regions. We sketched an efficient and scalable algorithm that discovers all maximal empty joins with a single scan over sorted two dimensional data set. We presented results from experiments performed on real data, showing the nature and quantity of empty joins that can occur in large, real databases. Knowledge of empty joins may be valuable in and of itself as it may reveal unknown correlations between data values. In this paper, we considered using this knowledge in query processing.

In the first technique we model the empty joins as materialized views, and so we exploit existing work on maintaining and using materialized views for query optimization. We presented experiments showing how the quality of optimization depends on the types and number of empty joins used in a rewrite.

In the second technique we use knowledge of empty joins for estimating join result size. Our approach provides a substantial improvement in the quality of estimates over *UDA*, a standard assumption in database systems. We also showed that our technique is superior to multidimensional histograms with respect to construction and maintenance.

References

[1] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *Proceedings SIGMOD, Philadelphia, Pennsylvania, USA*, pages 275–286, 1999.

[2] S. Agrawal, S. Chaudhuri, and V.R. Narasayya. Automated selection of materialized views and indexes in sql database. In *Proc. of VLDB*, pages 496–505, Cairo, Egypt, 2000.

[3] R. G. Bello, K. Dias, A. Downing, J. Feenan Jr., W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In *Proceedings of 24th VLDB*, pages 659–664, 1998.

[4] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *TODS*, 19(3):367–422, 1994.

[5] S. Ceri and J. Widom. Deriving production rules for constraint maintanance. In *Proceedings of the* 16^{th} *VLDB*, pages 577–589, Brisbane, Australia, 1990.

[6] U. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM TODS*, 15(2):162–207, June 1990.

[7] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the 11th ICDE*, pages 190–200, Taipei, Taiwan, 1995. IEEE Computer Society.

[8] I-Min. A. Chen and R. C. Lee. An approach to deriving object hierarchies from database schema and contents. In *Proceedings of the 6th ISMIS*, pages 112–121, Charlotte, NC, 1991.

[9] Q. Cheng, J. Gryz, F. Koo, C. Leung, L. Liu, X. Qian, and B. Schiefer. Implementation of two semantic query optimization techniques in DB2 UDB. In *Proc. of the 25th VLDB*, pages 687–698, Edinburgh, Scotland, 1999.

[10] W. Chu, R. C. Lee, and Q. Chen. Using type inference and induced rules to provide intensional answers. In *Proceedings of the 7th ICDE*, pages 396–403, Kobe, Japan, 1991.

[11] OLAP Council. APB-1 OLAP Benchmark Release II, November 1998. (www.olapcouncil.org).

[12] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of 22nd VLDB*, pages 330–341, Bombay, India, 1996. Morgan Kaufmann.

[13] J. Edmonds, J. Gryz, D. Liang, and R. J. Miller. Mining for empty rectangles in large data sets. In *Proceedings of the 8th ICDT*, pages 174–188, London, UK, 2001.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, New York, 1979.

[15] P. Godfrey, J. Gryz, and C. Zuzarte. Exploiting constraint-like data characterizations in query optimization. In *Proceedings of Sigmod*, pages 582–592, Santa Barbara, CA, 2001.

[16] J. Gryz, B. Schiefer, J. Zheng, and C. Zuzarte. Discovery and application of check constraints in DB2. In *Proceedings of ICDE*, pages 551–556, Heidelberg, Germany, 2001.

[17] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering Bulletin*, 18(2):3–18, 1995.

[18] M.T. Hammer and S.B. Zdonik. Knowledge-based query processing. *Proc. 6th VLDB*, pages 137–147, October 1980.

[19] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: An attribute-oriented approach. In *Proceedings of the 18th VLDB*, pages 547–559, Vancouver, Canada, 1992.

[20] C. N. Hsu and C. A. Knoblock. Using inductive learning to generate rules for semantic query optimization. In *Advances in Knowledge Discovery and Data Mining*, pages 425–445. AAAI/MIT Press, 1996.

[21] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Proceedings of the SIGMOD, San Jose, California*, pages 233–244, 1995.

[22] M. Jarke, J. Clifford, and Y. Vassiliou. An optimizing PROLOG front-end to a relational query system. In *SIGMOD*, pages 296–306, 1984.

[23] J.J. King. Quist: A system for semantic query optimization in relational databases. In *Proc. 7th VLDB*, pages 510–517, Cannes, France, September 1981.

[24] Ju-Hong Lee, Deok-Hwan Kim, and Chin-Wan Chung. Multi-dimensional selectivity estimation using compressed histogram information. In *Proceedings SIGMOD, Philadelphia, Pennsylvania, USA*, pages 205–214, 1999.

[25] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the 14th PODS*, pages 95–104, San Jose, California, 1995. ACM Press.

[26] M. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database. *ACM Computing Surveys*, 20(3):191–221, 1988.

[27] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings SIGMOD, Seattle, Washington, USA*, pages 448–459, 1998.

[28] M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of SIGMOD, Chicago, Illinois*, pages 28–36, 1988.

[29] A. Namaad, W. L. Hsu, and D. T. Lee. On the maximum empty rectangle problem. *Applied Discrete Mathematics*, (8):267–277, 1984.

[30] *SQL Reference Manual, Oracle 8i, Realease 8.1.5*. 500 Oracle Parkway, Redwood City, CA 94065, 1999.

[31] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of SIGMOD*, pages 294–305, Montreal, Canada, 1996.

[32] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB'97*, pages 486–495, 1997.

[33] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path election in a relational database management system. *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 23–34, May 1979.

[34] S. Shekar, B. Hamidzadeh, A. Kohli, and M. Coyle. Learning transformation rules for semantic query optimization. *TKDE*, 5(6):950–964, December 1993.

[35] S.T. Shenoy and Z.M. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):344–361, September 1989.

[36] M.D. Siegel. Automatic rule derivation for semantic query optimization. In *Proceedings of the 2nd International Conference on Expert Database Systems*, pages 371–386, Vienna, Virginia, 1988.

[37] D. Simmen, E. Shekita, and T. Malkems. Fundamental techniques for order optimization. In *Proceedings of SIGMOD*, pages 57–67, 1996.

[38] D. Srivastava, S. Dar, H.V. Jagadish, and A. Levy. Answering queries with aggregation using views. In *Proceedings of the 22nd VLDB*, pages 318–329, Bombay, India, 1996.

[39] Nitin Thaper, Sudipto Guha, Piotr Indyk, and Nick Koudas. Dynamic multidimensional histograms. In *Proceedings of SIGMOD*, pages 428–439, 2002.

[40] Transaction Processing Performance Council, 777 No. First Street, Suite 600, San Jose, CA 95112-6311, www.tpc.org. *TPC Benchmark*^{TM} *D*, 1.3.1 edition, February 1998.

[41] Clement T. Yu and Wei Sun. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):362–375, September 1989.

[42] D. Zilio, C. Zuzarte, S. Lightstone, W. Ma, G. Lohman, R. Cochrane, H. Pirahesh, L. Colby, J. Gryz, E. Alton, D. Liang, and G. Valentin. Recommending materialized views and indexes with ibm db2 design advisor. In *Proceedings of 1st International Conference on Autonomic Computing*, pages 180–188, New York, 2004.