Holes in Joins

Jarek Gryz and Dongming Liang

Technical Report CS-2001-03

June 18, 2001

Department of Computer Science

4700 Keele Street North York, Ontario M3J 1P3 Canada

Holes in Joins

Jarek Gryz and Dongming Liang
York University
Toronto

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 two real data sets. We then explore potential applications of this knowledge to query optimization. By modeling empty joins as materialized views, we show how knowledge of these regions can be used to improve query performance.

1 Introduction

A join of relations in real databases is usually much smaller than their cartesian product. For example, the OLAP Benchmark from [10] 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 milions 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 (which we call *empty joins* in this paper) be characterized in an interesting way? And secondly: Can this knowledge be useful in query processing? Consider the following example.

Example 1 *Consider* **Lineitem** *and* **Order** *tables in TPC-H [29]. The* **o_orderdate** *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. Assume that an item is always shipped within a year from the time it is ordered. This is depicted graphically in Figure 1. 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**. *The remaining portions of the tables will not appear together in the join result.*

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 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 [29].

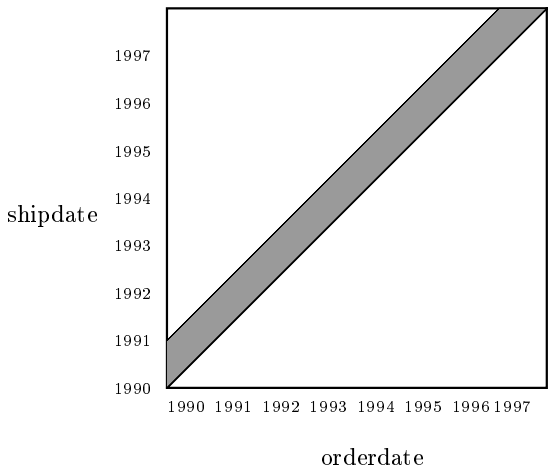

Figure 1: Distribution of tuples with respect to the values of **shipdate** and **orderdate**.

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 optimization. Consider the following example.

Example 2 *Consider the following query 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, this query can be rewritten as:

```
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.[1] *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.*

[1] We are assuming that the selection is done *before* the join execution.

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 [15] 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. Real data is likely to be distributed more randomly, yet not uniformly. In this paper, we propose 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 to not only to find empty joins in the data, but to fully characterize that empty space. Specifically, we discover the set of all maximal empty joins in a two dimensional data set (that is, a two-way join). 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 another 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. Indeed, we show that data skew - a curse of query optimization - can have a straightforward, practical application for that very query optimization.

In Section 2, we formally introduce this problem and sketch an algorithm (which appears originally in [12]) for finding the set of all maximal empty joins in a dataset. In Section 3, 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 4, we describe a technique illustrating how knowledge of empty joins can be used in query processing. We model the empty joins as materialized views, and so we exploit existing work on using and maintaining materialized views. Our solution therefore has the highly desirable property that it provides new optimization method without requiring any change to the underlying query optimization and processing engine. We also present experiments showing how the quality of optimization depends on the types and number of empty joins used in a rewrite. Related work is described in Section 5. Conclusions and future work are presented in Section 6.

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).[2]

[2]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.

2.1 Empty Join Representation

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 $\langle v_x, v_y \rangle$ 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 $\langle x, y \rangle$ of the matrix if and only if $\langle v_x, v_y \rangle \in D$ where v_x is the x^{th} smallest value in X and v_y the y^{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*[3] 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.*

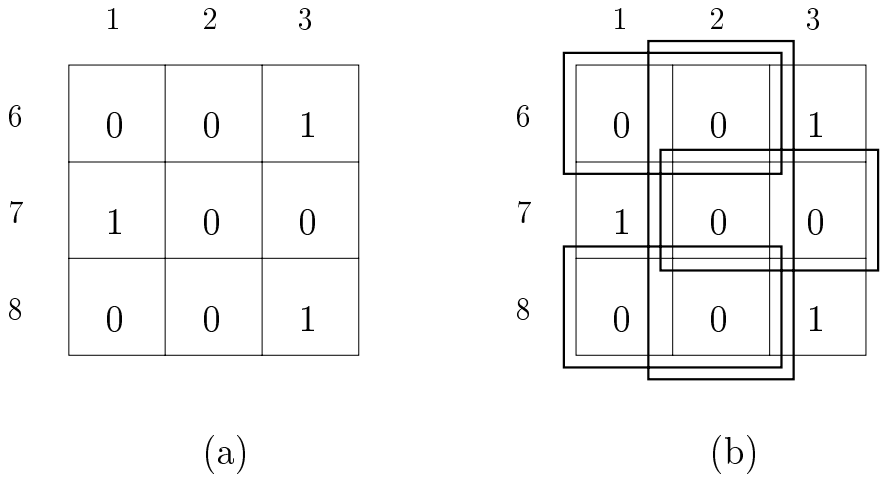

Figure 2: The matrix and some of the empty rectangles (marked with thick lines) for Example 3

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 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

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

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, [23] 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 [12] 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 [13].) This problem is NP-complete, and hence is impractical for use in large data sets. Besides, as we shall show in Section 4 that for the purpose of query optimization, it is more important to have large rectangles than to have the minimum number of rectangles.

2.2 Overview of the Algorithm

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 $\langle v_x, v_y \rangle$ is stored on disk 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 [12] for the details of the algorithm). We shall insure 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 stored on disk sorted with respect to the Y domain. Tuples from D are read sequentially off the disk in this sorted order. When the next tuple $\langle v_x, v_y \rangle \in D$ 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 [12]). However, in all experiments described in the remainder of this paper, we only consider two dimensional datasets. This is a consequence of the following result.

Theorem 2.1 *The number of maximal 0-hyper-rectangles in a d-dimensional matrix is* $\Theta(n^{2d-2})$, *for which* $n = |X| = |Y|$.

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 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, 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. For consiseness, we only present the results of five representative[4] tests here. For all reported tests the mining algorithm ran in less than 2 minutes.

Table 1 contains relevant information about the tables and attributes considered in the tests: the number of tuples in each of the joined tables $|R|$ and $|S|$, the size of the cartesian product of the tables $|R \times S|$, the size of the join $|R \bowtie S|$, and the number of distinct values of the two attributes of interest, $|X|$ and $|Y|$ (where X is the domain of A and Y is the domain of B).

Test	$\|R\|$	$\|S\|$	$\|R \times S\|$	$\|R \bowtie S\|$	$\|X\|$	$\|Y\|$
1	931,174	1,654,700	$1.5 * 10^{12}$	7,610,723	525	8
2	624,473	1,654,700	10^{12}	1,654,700	6	37,716
3	931,174	624,473	$.5 * 10^{12}$	907,736	525	423
4	931,174	1,654,700	$1.5 * 10^{12}$	7,610,723	47	7
5	624,473	1,654,700	10^{12}	1,654,700	5	38,203

Table 1: Input Data

Table 2 contains the mining results: the number of discovered empty joins E and the sizes of the 5 largest empty joins measured by two different metrics S_A and S_T. The first metric 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 $(x_0, x_1), (y_0, y_1)$ over attributes A and B with domains X and Y respectively in tables R and S respectively. The relative size of the join with respect to the covered area, $S_A(E)$, is defined as:

$$S_A(E) = \frac{(x_1 - x_0) * (y_1 - y_0)}{[max(X) - min(X)] * [max(Y) - min(Y)]} * 100\% \quad (1)$$

[4]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	E	Size of largest 5 empty joins measured by the two metrics									
		S_A					S_T				
1	269	74	73	69	7	7	.02	.016	.008	.008	.004
2	29,323	68	58	40	37	28	.05	.03	.03	.03	.03
3	13,850	91.6	91.6	91.3	91.3	83.1	1.85	1.84	1.11	.03	.03
4	7	8.8	2.1	1.2	0.6	0.3	9.8	5.4	3.9	2.4	$1*10^{-8}$
5	25,307	39.9	39.8	24	20	20	50	6	4.4	3.9	2.2

Table 2: Number and Sizes of Empty Joins

The reason we define a second metric, $S_T(E)$, is that the first metric, $S_A(E)$, can be misleading as a measure of *empty joins*. What we discovered was that some of the "empty" joins were empty, because there were no tuples falling in one or both of the respective ranges of the two attributes, and *not* beacase the tuples would not join. Measuring area using S_A measures how many *domain* values do not participate in the join. However, an empty join might exist solely because *no* tuples in R (or S) exist for *those* domain values in A (or B). Hence, we devise a more meaningful ranking of the empty joins. We measure now how many contiguous (with respect to the domains of each of the attributes) *tuples* in essence do not join. From this perspective, the "size" of an empty join is defined as the product of the number of tuples within the ranges of the two attributes relative to the size of the cartesian product of the two tables. It is defined formally as follows.

Let $|R_{x_0}^{x_1}|$ be the number of tuples in the range (x_0, x_1) of attribute A, and $|S_{y_0}^{y_1}|$ be the number of tuples in the range (y_0, y_1) of attribute B. Then the relative size of the empty join E with respect to the number of tuples in the cartesian product of the two tables R and S is defined as:

$$S_T(E) = \frac{|R_{x_0}^{x_1} \times S_{y_1}^{y_0}|}{|R \times S|} * 100\% \tag{2}$$

We note that for uniform data distribution with sufficiently dense attribute domains,[5] S_T and S_A yield identical results.

We make the following observations:[6]

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 [12].

2. In virtually all tests, extremely large empty joins (measured by S_A) 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.

[5] More precisely: S_T and S_A yield identical results for an empty join $(x_0, x_1), (y_0, y_1)$ if there is at least one tuple in the range (x_0, x_1) if attribute A and at least one tuple in the range (y_0, y_1) of attribute B in uniformly distributed data.

[6] Due to confidentiality of the data we are unable, unfortunately, to describe the meaning of some of the discovered empty joins.

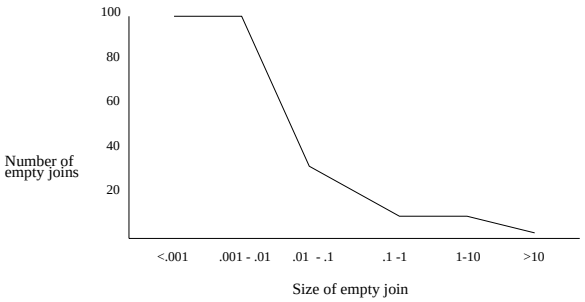

Figure 3: Distribution (with respect to S_A) of empty joins in Test 1.

3. The empty joins overlap substantially. The five S_A-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 a large area of the join matrix, that is, the combination of values from the domains of the two attributes (see Figure 4).

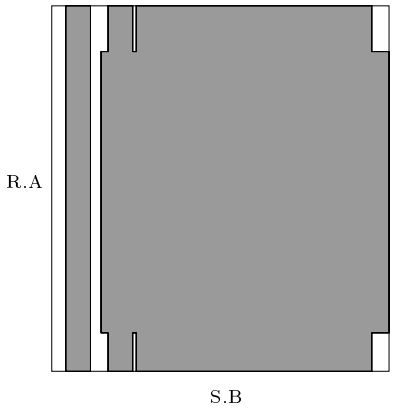

Figure 4: The area (in dark) covered by thr five S_A-largest empty joins in Test 1 indicates no tuples in the join for the respective ranges of attributes A and B.

4. The metrics S_A and S_T lead to very different rankings of empty joins. Indeed, none the S_A-largest join is among the ten S_T-largest joins in any of the experiments. For example, for Test 1, the S_A-largest empty join is S_T-ranked at 250-th; the S_T-largest is 267-th in S_A-ranking.

5. The "sizes" of the S_T-largest empty joins appear as much smaller than the S_A-largest joins. Some of them, however, can be objectively large as well (see Test 5 and Figure 5). This is important, as the quality of optimization achieved by the use of empty joins is determined primarily by the number of tuples removed from the tables before the join execution.

6. The experiments reveal two types of data skew. The first one is non-uniformity of data distribution within a domain of one (or both) attribute. This is indicated by S_A-large empty joins without corresponding S_T empty joins (that is, the same empty join measured by the S_T metric is of size zero) for the same range of attributes. Consider an empty join E with

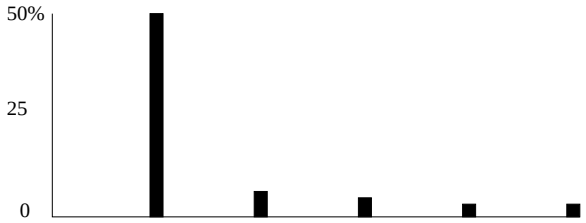

Figure 5: The sizes (with respect to S_T) of the largest empty joins in Test 5.

the coordinates $(x_0, x_1), (y_0, y_1)$. If no empty joins have been discovered for this combination of the ranges of the attributes using metric S_T, then there are no tuples in the range (x_0, x_1) or (y_0, y_1).

The second type of data skew is non-uniform distribution of the values of the two attributes with respect to each other in the universal relation. The very existence of S_T-large empty joins means that there are ranges of each attribute with many tuples in the respective two tables, yet no tuples in the join.

4 Using Empty Joins in Query Optimization

4.1 Query Rewriting

Using the algorithm of the previous section, we can discover empty joins in any two dimensional dataset. We now turn to the question of how to use this knowledge effectively 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.

Let Q be a SELECT-FROM-WHERE query representing a two-way join with two projected attributes X and Y.

select X, Y
from R_1, R_2
where JoinCond(Q)

Suppose we have mined the result of this query for empty joins and determined that the region $(x_0 \leq X \leq x_1,\ y_0 \leq Y \leq y_1)$ is empty. We model this region using the following view.

create view empty as
select *
from R_1, R_2
where JoinCond(Q)
and X between x_0 **and** x_1
and Y between y_0 **and** y_1

We can now use existing results on determining whether a view can be used to answer a query and on rewriting queries using such views [28]. Rather than restating these results, we present an example of how they would be used in our context.

Example 4 *Suppose that we mined for empty joins in a join represented by the following query:*

J: select *o_orderdate, l_extendedprice,*
from *Lineitem l, Order o*
where *l_orderkey = o_orderkey*

Suppose we detected that there are no items priced over 1,000,000 ordered before 01.01.95. We represent this information in the following view.[7]

V: create view *empty* as
select *
from *Lineitem l, Order o*
where *l_orderkey = o_orderkey*
AND l_extendedprice > 1,000,000
AND o_orderdate < '1995.01.01'

Now, consider a variant of the query of Example 2.

Q: select *
from *Lineitem l, Order o*
where *l_orderkey = o_orderkey*
AND o_orderdate BETWEEN '1985.01.01' AND '1996.01.01'
AND l_extendedprice > 1,000,000

Using the rewrite algorithm of [28], we can rewrite Q as Q' which uses V.

Q': select *
from *Lineitem l, Order o*
where *l_orderkey = o_orderkey*
AND o_orderdate BETWEEN '1995.01.01' AND '1996.01.01'
AND l_extendedprice > 1,000,000
union
select *
from *empty*

[7] Note that the missing endpoints of the ranges are implicit as the maximum and the minimum value of *l_extendedprice* and *o_orderdate* respectively.

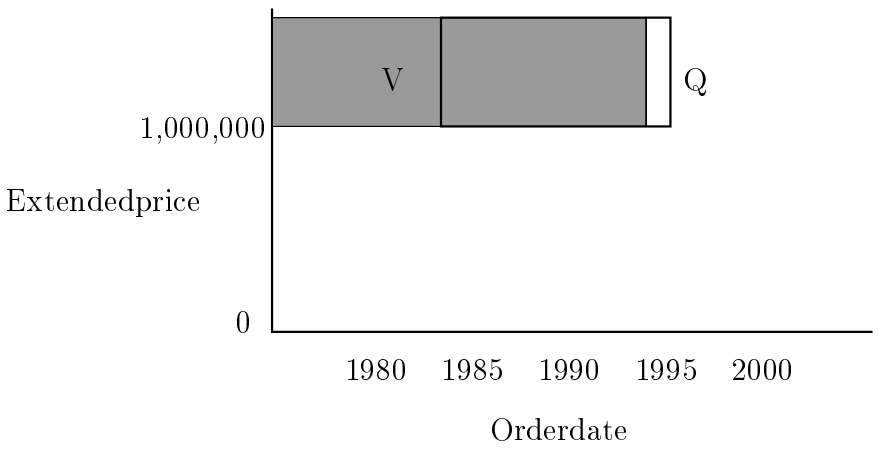

Figure 6: Query Q overlaps the empty View V.

Given that the view is empty the second block of Q' will also be empty. What we have done is notice that the Query Q and the View V overlap as depicted in Figure 6. Using this information, we further rewrite the query by dropping the second block to obtain simply the following:

```
Q'':  select   *
      from     Lineitem l, Order o
      where    l_orderkey = o_orderkey
               AND o_orderdate BETWEEN '1995.01.01' AND '1996.01.01'
               AND l_extendedprice > 1,000,000
```

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.

4.2 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 [22, 28]. 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 6, 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* [11]. Consider Figure 7, 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.

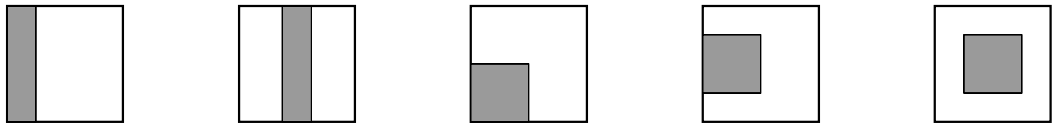

Figure 7: Overlaps of increasing complexity between the query and empty joins.

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 [6] 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 9 is guaranteed not to produce worse performance than in the original query provided this rewrite can be found efficiently. We believe that other cases of this type can be identified.

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.

4.3 Quality of 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.[8] No indexes have been created or used.

[8]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.

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 $*$
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 8.

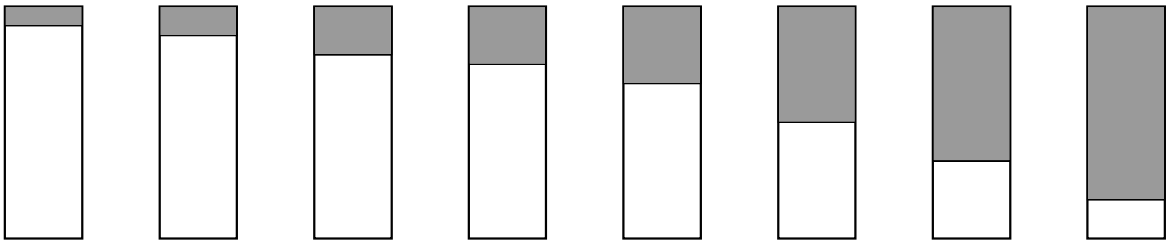

Figure 8: Increasing overlaps between a query and an empty join.

Experiment	1	2	3	4	5	6	7	8
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 3: 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 9.

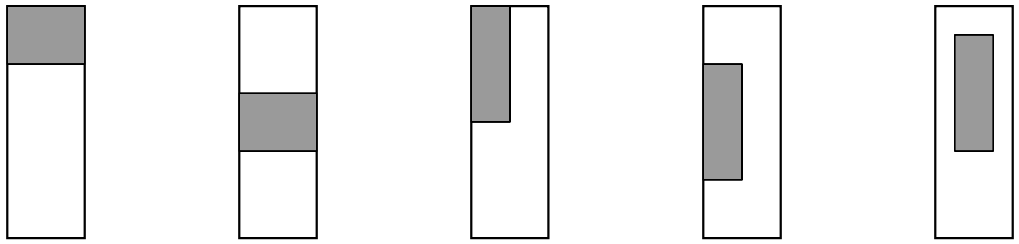

Figure 9: Overlaps of increasing complexity between the query and empty joins.

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

Table 4: Impact of the type of the overlap used in rewrite on query performance.

As shown in Table 4, only the first two types of the overlap provide substantial performace 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:

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 ans 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 10. The results are shown in Table 5.

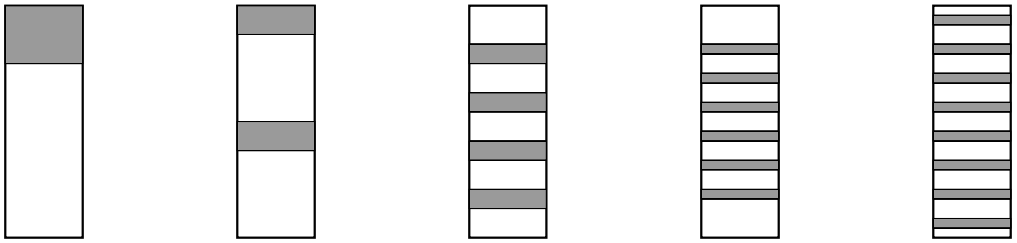

Figure 10: Overlaps with increasing number of empty joins.

Experiment	a	b	c	d	e
Reduction of execution time (%)	39	38.4	38.3	35.9	34.3

Table 5: Impact of the type 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.

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** $5,550$)]

4.4 Selection and Maintenance of Empty Joins

Since the number of empty joins discovered in real datasets is large, they cannot all be maintained. The decision on which joins to maintain depends primarily on the stability of the query workload and the frequency of updates. In an environment with a stable workload of queries and frequent updates, only a few empty joins which provide the best optimization for the workload queries should be kept. Since empty joins can be modeled as materialized views, 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 [1, 16, 2].

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 [16] can be applied here as well. Since empty joins are a special case of matrialized views, 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 [24, 14] 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.

5 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 [4, 3, 30] or can be discovered from database content using machine learning or data mining approaches [7, 9, 18, 25, 27, 30]. It has also been suggested that such rules be used for query optimization [19, 25, 27, 30]. 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 [16, 22, 6, 28].

Also, since empty regions describe semantic regularities in data, they are similar to integrity constraints [14]. 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 [5, 17, 20, 21, 26, 8], which uses integrity constraints for that purpose.

6 Conclusions and Future Work

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

We are currently working on another way of modelling empty joins, this time as *statistical soft constraints* [14]. A statistical soft constraint is a constraint statement which is valid - with some error allowed - with respect to the *current* state of the database. We showed that soft constraints can be useful for better cardinality estimation as they reveal regularities within data that may be unknown to the optimizer.

References

[1] 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.

[2] Randall G. Bello, Karl Dias, Alan Downing, James Feenan Jr., William D. Norcott, Harry Sun, Andrew Witkowski, and Mohamed Ziauddin. Materialized views in Oracle. In *Proceedings of 24rd VLDB*, pages 659–664. Morgan Kaufmann, 1998.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

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

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

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

[18] 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.

[19] 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.

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

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

[22] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS*, pages 95–104, San Jose, California, 1995. ACM Press.

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

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

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

[26] 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.

[27] 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.

[28] D. Srivastava, S. Dar, H.V. Jagadish, and A. Levy. Answering queries with aggregation using views. In *Proc. of VLDB*, 1996.

[29] 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.

[30] 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.