Efficient Processing of Relational Queries with Constraints Over the Sum of Multiple Attributes

Chuang Liu¹  Svetlozar Nestorov¹  Ian Foster¹,²

¹University of Chicago, 1100 E. 58th Street, Chicago, IL, 60615
²MCS Division, Argonne National Laboratory, Argonne, IL 60439
{Chliu, evtimov, foster} @cs.uchicago.edu

Abstract

We identify and study an important class of relational queries involving constraints over the sum of multiple attributes (sum constraint queries). Finding all or a given number of results of these queries requires expensive join operations. These joins, in the absence of any other join conditions, effectively become cartesian products.

We develop rewriting techniques to rewrite a sum constraint query in order to enable its efficient processing by conventional relational database engines. Experimental results show that query rewriting achieves notable performance improvement for sum constraint queries without modifying database search engine.

For queries asking for a given number of results, we propose a ranking algorithm to order tuples based on their probability to satisfy all sum constraints in a query. We compare it with traditional ranking algorithms that rank tuples based on value of one attribute, and show that our method is more stable and efficient to handle sum constraint queries.

We also study a special but common type of sum constraint queries: self-join of a relation with symmetric sum constraints as join conditions. Considering the large number of possible execution plans, we prove that left-deep tree is always the best execution plan for this type of queries.

1 Introduction

As database technologies are applied in ever more diverse applications, we encounter the need to perform combinatorial searches on relational database. These combinatorial queries usually contain condition and (or) ranking criterion expressed as monotone functions on multiple attributes from these relations. To illustrate this type queries, consider the following simple example.

Example 1. Consider a database storing nutritional information for single serving of different kinds of food in the following 4 relations: Meats, Vegetable, Fruits, and Beverages. Suppose that a meal consists of single servings of each of the four kinds of food. We are interested in finding meals that satisfy various nutritional requirements, such that restrictions on the number of calories, grams of saturated fat, and amount of Vitamin C. For example, the daily USDA recommendations for a 30-year old female, who is moderately active, are 1800-2200 calories, less than 18g of saturated fat and 300mg of cholesterol, and at least 4mg of Vitamin B6 and 96mg of Vitamin C. Assuming that a main meal carries about half of the daily nutritional values, we can find all meals with the following SQL query:

```
SELECT M.name, V.name, B.name, F.name
FROM Meats AS M, Vegetables AS V, Beverages AS B, Fruits AS F
WHERE M.cal + V.cal + B.cal + F.cal > 900
AND M.cal + V.cal + B.cal + F.cal < 1100
AND M.Vb6 + V.Vb6 + B.Vb6 + F.Vb6 > 2
AND M.Vc + V.Vc + B.Vc + F.Vc > 38
AND M.fat + V.fat + B.fat + F.fat < 9
AND M.chol + V.chol + B.chol + F.chol < 150
```

Figure 1. An example query

Note that the query condition is a conjunction of six constraints over the sum of five attributes from four relations. This type of queries are commonly encountered in the context of document retrieval [1], multimedia data retrieval [2], geographic information system [3, 4], e-commerce [5], dynamic resource location in an Internet environment [6], and supply chain management. For example, in the context of supply chain management, a product supplier may control the total cost under a budget by carefully combining different service providers involved in producing, shipping and distributing products to construct a supply chain. Guha [7] shows that the decision-making for supply chain management can be implemented as queries for a set of tuples with constraints over aggregate of attributes values. In the context of Internet computing [8], applications need to find a set of
computation resources with desired total memory size and CPU speed to run in order to get good performance and high efficiency. Assuming computation resources are stored in relations, Liu [9] modeled resource location as relational database queries seeking to identify tuples from relations with constraint over the sum of attribute values. We define Sum Constraint Query to represent this type of queries.

Definition 1 (Sum Constraint Query) Sum Constraint is a constraint over the sum of multiple attributes. A sum constraint query is a query containing multiple sum constraints in the query condition.

Queries involving this type of query conditions have not been implemented efficiently by the conventional approach of composing pair-wise joins as each condition refers to more than two attributes in different relations. Because current join operators cannot test the satisfiability of an intermediate result until values of all attributes in a constraint are determined, the query evaluation process may produce a lot of intermediate results, which may lead to high memory and computation costs.

Previous works improve query processing efficiency by either extending current data search engine (e.g. introduce new search algorithms [2] and new join operators [1, 9]), or uses an sophisticatedly built index [7] which may cause extra cost to maintain for dynamic data. Thus the question that we address in this paper: can we introduce efficient support for such query condition into relational database systems without requiring significant modifications to the underlying database engine and building extra index structure? Please note that, my approach is not replacement to previous methods. Instead, this method can be use together with previous methods to allow more efficient execution of combinatorial queries on relational database systems.

In this paper, we introduce a new approach to enable sum constraint queries to efficiently execute by traditional relation database engine. We make the following contributions.

- We introduce a query rewriting technique that creates new query conditions that can be used by the current join operator to remove intermediate results that do not lead to any results at the early stage of execution. Our experimental results show that this technique achieves significant reductions in query response time.
- We propose a ranking-based algorithm searching for a given number of results. This ranking algorithm orders tuples based on their probability to satisfy all sum constraints in a query. We compare it with traditional ranking algorithms that rank tuples based on one attribute, and show that our method is more stable and efficient to handle sum constraint queries.
- We also study possible execution plans for a special type of Sum Constraint Queries in which join operations are on the same relation. The result shows that left-deep tree is the best plan.

The paper is organized as follows. In section 2, we review related work. We introduce a technique to rewrite a sum constraint queries into a query that can be more efficiently processed by existing database search engine in section 3. In section 4, we propose a ranking-based algorithm searching for a given K results. In section 5, we study the optimal execution plan for self-join queries with sum constraints. We evaluate the performance of our method in section 6, and summarize our work in section 7.

2 Related Work

The problem of processing sum constraint queries is closely related to the multi-join problem. A multi-join operation combines information from multiple relations to determine if an aggregation of tuples from those relations satisfies search conditions. Multi-joins are usually implemented by a set of pair-wise operations [10]. This method is efficient when the join condition consists of only equality or inequality comparisons of attributes from two join relations, as each individual join can eliminate tuples that do not satisfy its condition. Algorithms have been proposed for ordering pair-wise joins so as to obtain a minimal search space [11, 12]. (It is also possible to speed up queries by using multiple processors to parallelize the query processing [13, 14].) However, this pair-wise strategy does not work well for sum constraint queries. Because a sum constraint involves tuples from multiple relations, a pair-wise join operator cannot test the satisfiability of intermediate results based on a sum constraint until all attributes in the constraint are determined. Thus, a purely pair-wise strategy may generate many intermediate results that cannot lead to any solution.

Guha et al. [7] and Agarwal et al [15] address queries with aggregation constraints by building a sophisticated index. This index can be used to return results (approximate results in [7] or precise results in [15]) for sum constraint queries efficiently. As creating the index needs some complex computations (e.g. solving a series of knapsack problems), these approaches may cause extra load for database systems to maintain the index on dynamic data. Also, their works focus on seeking for tuples from the same relation with constraints involving two or three attributes. Our approach, instead, can be applied to combinatorial queries for tuples from multiple different relations.

Algorithms proposed by Fagin et al. [2] and Ilyas et al. [1] for top-k queries can be extended to implement queries with a constraint on the value of a monotone function, such as $A.attribute_1 + B.attribute_2 + C.attribute_3 > N$. They sort all join relations based on the value of attributes in the constraint, and then check the combination of tuples in the order of the value of the monotone function. However, this method can only use one constraint to
guide the search process, which is not efficient when there are multiple constraints. In this paper, we introduce an approach using all constraints to guide the search process, and show that our method can be integrated with existing algorithms to further improve the search performance.

Combinatorial search problem is a classical problem and has been widely studied in areas such as artificial intelligence, operations research, job scheduling, etc. Constraint programming [16] and mathematical programming [17] have been developed to solve this problem. Liu et al. [9] integrated constraint-programming techniques with traditional database techniques to solve sum constraint queries by modifying existing nest-loop join operators. In this work, we revisit this method and implement it without modifying existing database engines.

3 Query Rewriting

In this section, we show how we can rewrite a sum constraint query to allow more efficiently execution of this query on database search engine.

3.1 Motivation

Current database systems use multiple nested-loop operators to implement a sum constraint query. As an example, Figure 2 (a) shows the plan to execute the sum constraint query in Figure 1 (We assume in that figure that the execution plan is pipelining.) The problem of this execution plan is that join operators cannot remove intermediate results based on a sum constraint until value execution plan is that join operators cannot remove intermediate results. Therefore, except the top-level join operator, join operators will calculate the Cartesian product of involved relations.

![Figure 2. (a) Conventional and (b) improved execution plans for a sum constraint query](image)

To solve this problem, we want an execution plan that is able to filter tuples and remove intermediate results based on sum constraints in the query condition. As shown in Figure 2(b), this new plan differs from the original in two respects. First, we add selection operators that filter tuples that cannot lead to a solution. Second, we introduce new query constraints that can be used by join operators to generate only intermediate results that possibly lead to a query result. Because every intermediate result will join tuples from other relations in the following join operators, this new plan saves computation time, I/O, and memory needed by the following join operations.

Although this execution plan is straightforward, it needs more intelligence in the database query processor to determine the selection conditions in the selection operators and to create constraints for join operators to filter intermediate results.

3.2 Query Rewriting

The essence of our method is rewriting each sum constraint in the query condition into a set of simpler constraints that can be used to create the execution plan shown in Figure 2(b). In other words, we need provide selection operator constraints to filter tuples, and provide join operator constraints to remove intermediate results.

3.2.1 Selection operators

Selection operators use range constraint on single attribute to filter tuples from relations. We present the method to rewrite a sum constraint into a set of range constraints. A sum constraint has the following general form.

$$D_1 + \ldots + D_n \sim C$$  \hspace{1cm} (3.1)

Here, $C$ is a constant, $D_i$ ($i = 1..n$) are attributes that appear in this constraint, and $\sim$ represents a logic operator that can be ‘$>$’, ‘$<$’, ‘$\geq$’, and ‘$\leq$’. This constraint can be represented as follows.

$$D_k \sim C - \sum_{i=1, i \neq k}^n D_i$$  \hspace{1cm} (3.2)

We can calculate the bounds of the expression on the right side of this formula as follows.

$$L_k = C - \sum_{i=1, i \neq k}^n \text{Max}(D_i)$$

$$H_k = C - \sum_{i=1, i \neq k}^n \text{Min}(D_i)$$

Here, Max(D_i) and Min(D_i) respectively denote the upper and lower bounds of the attribute $D_i$ in related relation, and $H_k$ and $L_k$ are the upper and lower bounds of the expression on the right side of formula 3.2. These values can be used to create a range constraint on attributes on the left side of formula 3.2. If $\sim$ is the logic operator ‘$<$’ or ‘$\leq$’, then we create a range constraint on $D_k$:

$$D_k \sim H_k$$  \hspace{1cm} (3.3)

If $\sim$ is the logic operator ‘$>$’ or ‘$\geq$’, then the range of $D_k$ is:

$$D_k \sim L_k$$  \hspace{1cm} (3.4)
Any value not in this range cannot possibly satisfy the sum constraint shown in formula 3.1. Through formula 3.3 and 3.4, we obtain a range for all attributes in the constraint. Selection operators (as shown in Figure 2(b)) can use these ranges to filter tuples that will not lead to any query results.

### 3.2.2 Join operators

Even with these selection operators, the execution of constraint query is still very expensive because it needs to calculate the Cartesian product of involved relations after being processed by selection operators. Another possible improvement is to use join operators to remove intermediate results not leading to any query results. However, current join operators cannot use sum constraints to remove an intermediate result until values of attributes involved in a constraint are determined.

To this end, we use a sum constraint to create for each join operator new constraints that contain only attributes appeared in intermediate results produced by this join operator. Because values of involved attributes in these new constraints are determined in an intermediate result, join operators can evaluate these constraints and remove results that do not satisfy these constraints. For the example in Figure 2, we create, for the join operator between relation M and F, constraints containing attributes from these two relations; and for the join operators between relation B and the join result of M and F, constraints containing attributes from relation M, F, and B.

We show in the following how to create new constraints for a join operator whose intermediate results contain attributes D, j (j=1...k). First, we rewrite the sum constraint (shown in formula 3.1) by moving attributes not in D, j (j = 1...k) to the right side of the formula. The result is shown in formula 3.5.

$$\sum_{j=1}^{k} D_j - C - \sum_{i=k+1}^{n} D_i$$

for k=1 ... n  \hspace{1cm} (3.5)

The left side of formula is the sum of k attributes. We can calculate the bounds (H, k and L, k as upper and low bound respectively) of the expression on the right side of this formula.

$$L_k = C - \sum_{i=k+1}^{n} \text{Max}(D_i)$$

$$H_k = C - \sum_{i=k+1}^{n} \text{Min}(D_i)$$

Then, we can create constraints on attribute D, j (j = 1...k) as follows.

$$\sum_{j=1}^{k} D_j - H_k$$

If ~ is ‘<’ or ‘\leq’  \hspace{1cm} (3.6)

The constraints contain only attribute D, j (j=1...k), and can be used by the targeted join operator to remove intermediate results. Using this rewriting technique, we rewrite the query in Figure 1, and show part of the rewritten query in Figure 3.

```sql
SELECT M.name, V.name, B.name, F.name 
FROM Meats AS M, Vegetables AS V, Beverages AS B, Fruits AS F 
... 
```

**Figure 3. An example of query rewriting**

In Figure 3, we only show the new created constraints from the two sum constraints on attribute Cal due to the limitation of the space. Besides original sum constraints, the new query contains for each table range constraints on attribute Cal that can be used by selection operators to filter tuples, and sum constraints containing attributes from two and three relations, which can be used by the join operator between M and F, and the join operator between M, F, and B to remove intermediate results.

When rewriting a query, we assume we already know the order of the join operators. To decide the order to join, we calculate the number of tuples in each relation after filtered by selection operators, and decide the join order based on the size of filtered relations [10].

In summary, we rewrite a sum constraint into a set of simpler constraints that can utilize by selection operators and join operators to improve query performance. Although this rewriting technique adds more constraints (2*n-1 new constraints for a join operation between n relations) into the query condition that may cause some extra computations to validate a result, this cost is trivial comparing to gains archived by reducing tuple reading operations and reducing the number of the intermediate results that need to create and verify. We run a benchmark to quantify the performance improvement in section 6. Comparing to previous method proposed in [1, 2, 7, 15], rewriting technique proposed in this report does not need to modify the current database engine or build complex index structure, and can be easily deployed on current database systems.
Many applications are only interested in a given number K of results for a sum constraint query. For example, we want to find 50000 meals with different combinations of fruits, beverages, meats, and vegetables. Please note that we focus on finding any K results instead of top-K results in this paper.

Previous work by Ilyas et. al. [1] and Fagin et. al. [2] can be used to find K results satisfying a sum constraint efficiently. For the query in Figure 1, we sort tuples in relation Fruits, Meats, Vegetable, and Beverages based on attribute Vc. By combining tuples with higher value of Vc first, the search engine can efficiently find K meals with total number of Vc more than the requirement specified in the constraint. For convenience, we call tuples with higher value of Vc as high quality tuples for this constraint.

However, a sum constraint query may contain multiple sum constraints over different attributes, and these constraints may ‘conflict’ with each other. For example, the query in Figure 1 contains a constraint requiring the total calories more than 900, which favors tuples with higher calories; and a constraint requiring total vitamin C more than 38, which favors tuples with higher Vitamin C. These two constraints ‘conflict’ because foods containing higher vitamin C usually have lower calories. Therefore, it might be better to search for meals with medium number of calories or vitamin C first in order to find meals satisfying both constraints.

Instead of ordering tuples based on its quality to just one constraint, we propose a ranking method to order tuples based its quality to satisfy all constraints. First, we define quality of a tuple to a constraint as follows. For a constraint C requiring the sum of attribute attr more than a given value B, we define the quality of a tuple T to this constraint as:

\[
\text{Quality}(T, C) = \begin{cases} 1 & \text{if } T.\text{attr} > B \\ T.\text{attr}/B & \text{otherwise} \end{cases}
\]

Here we use T.\text{attr} to represent the value of attribute \text{attr} in tuple T. The quality is a value between 0 and 1, and can be understood as the degree of this tuple satisfying this constraint. If the attribute value is more than B, this tuple fully satisfies this constraint. We represent its quality as value 1. If attribute value is less than B, this tuple satisfies T.\text{attr}/B percentage of the total requirement.

For a constraint C requiring the sum of attribute attr less than a given value B, we define the quality of a tuple T to this constraint as:

\[
\text{Quality}(T, C) = \begin{cases} \text{-MAX_INT} & \text{if } T.\text{attr} > B \\ -T.\text{attr}/B & \text{otherwise} \end{cases}
\]

Because the constraint requiring the sum of attribute value less than a value B, each tuple with positive attribute contribute negatively to the satisfaction of this constraint. If the attribute value is more than B, this tuple cannot satisfy this constraint assuming all attribute are positive. We represent its contribution as a minimal value.

If attribute value is less than B, this tuple uses up T.\text{attr}/B percentage range specified in the constraint.

By summing up the quality of each tuple to each constraint, we get the quality of each tuple to final query results. Also, we would like to give preference to tuples with higher quality to constraints that is hard to satisfy.

We evaluate the hardness of a constraint based on the statistics of attribute values in relation. We use the example in Figure 1 to illustrate our method. In Table 1, we show average values of attributes contained in sum constraints. For example, each serve of fruit has an average of 86 K-calories, 0.07mg vitamin B6, 31mg vitamin C, 0.07g saturated fats, and none cholesterol as shown in the first row of the table.

### Table 1. Average nutrition value of foods

<table>
<thead>
<tr>
<th></th>
<th>Cal</th>
<th>Vb6</th>
<th>Vc</th>
<th>fa_sat</th>
<th>chol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fruits</td>
<td>86</td>
<td>0.07</td>
<td>31</td>
<td>0.07</td>
<td>0</td>
</tr>
<tr>
<td>Meats</td>
<td>221</td>
<td>0.37</td>
<td>1.16</td>
<td>4.95</td>
<td>105</td>
</tr>
<tr>
<td>Vegetables</td>
<td>63</td>
<td>0.18</td>
<td>26.4</td>
<td>0.23</td>
<td>0.69</td>
</tr>
<tr>
<td>Beverage</td>
<td>142</td>
<td>0.22</td>
<td>67.6</td>
<td>0.7</td>
<td>1.46</td>
</tr>
</tbody>
</table>

We represent the hardness of a constraint by H. For a constraint C requiring the sum of attribute \text{attr} from different relations more than a given value B, we define H as the ratio between the sum of average value of these attributes and boundary B.

\[
\text{H} = \text{sum(}\text{avg}(T.\text{attr})) / B
\]

For a constraint requiring the sum of attributes less than B, we define H as

\[
\text{H} = B / \text{sum(}\text{avg}(T.\text{attr}))
\]

Assume most of tuples’ attribute values are around its average value. Therefore, the bigger H is, the more the number of tuples that can satisfy this constraint.

We show in Table 2 six constraints’ H values for the query in Figure 1. The first row in Table 2 shows six constraints in the query. We use +N to represent a constraint requiring the sum of the related attribute more than N. For example, +900 in this table represents a constraint requiring the sum of attribute Cal more than 900. The second row in the table shows the hardness of each constraint. From these values, we can see that the sum constraint on Vb6 is the hardest one to satisfy.

### Table 2. Selectivity of six sum constraints on nutritional facts of a meal. C represents a constraint, and H represents the hardness of a constraint.

<table>
<thead>
<tr>
<th></th>
<th>cal</th>
<th>Vb6</th>
<th>Vc</th>
<th>fa_sat</th>
<th>chol</th>
<th>cal</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>+900</td>
<td>+2</td>
<td>+38</td>
<td>-9</td>
<td>-150</td>
<td>+1100</td>
</tr>
<tr>
<td>H</td>
<td>0.57</td>
<td>0.42</td>
<td>3.32</td>
<td>1.5</td>
<td>1.4</td>
<td>2.15</td>
</tr>
</tbody>
</table>

We give higher weight to a tuple with higher quality to harder constraints by calculating its contribution by the following formula.
Quality(T) = \sum_{i=1}^{N} \frac{Quality(T, C_i)}{H(C_i)}

We sort tuples in relations according to their quality to all sum constraints. Then, we join tuples with higher quality first. In this way, we check combinations of tuples that most likely to satisfy all sum constraints in a query first.

5 Execution Plan of Self-Join

A special but important case of sum constraint query is that the join operation is on the same relation. For example, in the context of parallel computing, a query for “eight computers with total memory more than 32 G and total cost less than $1000” is implemented by the self-join of the relation ‘computer’ that stores computer information. For convenient, we call it a \textit{k-time self-join} when a relation joins to itself \textit{k} times.

Database traditionally chooses left-deep tree to execute multi-join queries. However, an interesting observation is that we can reduce the number of join operations for a \textit{k}-time self-join by joining intermediate results directly. For example, for a 4-time self-join of relation \(A\), we can join the relation with itself first to get the result of 2-time self-join. Instead of joining this result with the relation \(A\) as left-deep tree does, we can join the result of 2-time self-join with itself to get the result of 4-time self-join. We show this plan and left-deep tree in Figure 4. For convenience, we name the result of 2-time self-join of \(A\) as \(A_2\). This graph shows that new plan needs only two join operations instead of three in the left-deep tree.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure4.png}
\caption{Two possible execution plans for the 4-times self-join of a relation}
\end{figure}

For a \textit{k}-time self-join, there are many possible ways to build a new plan that contains less join operations than left-deep tree does. Now the question is: do these new plans outperform the tree-deep tree? And how to choose the best plan for \textit{k}-time self-join?

5.1 Performance Metrics

To compare execution plans, we use the number of I/O operations as the performance metric. Before introducing the calculation method, we define terms used in the left part of the paper as follows:

- \(A_i\): represents the result after \(i\)-time self-join of relation \(A\).
- \(|R|\): represent the size of a relation \(R\).
- \(J(L, R)\): represents the cost of joining relation \(L\) and \(R\). \(L\) and \(R\) represent the left and right argument of a join operator respectively.
- \(C(P)\): represents the I/O operations caused by a plan \(P\).

For simplicity, we assume join operators are nest-loop join operators and every tuple read will cause one I/O operation. Based on these assumptions, we calculate the number of I/O operations caused by a join operation between relation \(L\) and \(R\):

\[ J(L, R) = |L| + |L| \times |R| \]  

We can calculate the I/O operation of an execution plan \(P\) by summing up the I/O operations caused by each join operators in this plan. For example, we can calculate the number of I/O operations caused by the left-deep tree in Figure 4:

\[ C(\text{left-deep tree}) = \sum_{i=1}^{3} (|A_i| + |A_i| \times |A_i|) \]

\(A_i\) represents the intermediate results after \(i\) self-join. This formula adds up the number of I/O operations for each of seven join operators.

5.2 Best plan for \textit{k}-time self-join queries

It is a challenge to find the best plan for a \textit{k}-time self-join query because there exists a large number of possible execution plans to evaluate this query. In this section, we introduce an algorithm to find the best execution plan for a \textit{k}-time self-join query. Using this algorithm, we prove that left-deep tree is always the best plan for sum constraint queries. We define terms used in the algorithm as follows:

- \(C_b(k)\): represents the cost of the best plan for \textit{k}-time self-join.

A \textit{k}-time join can be calculated by the joining intermediate result \(A_{k,j}\) with \(A_j\) for \(j=1\ldots k\). The cost is

\[ \text{Cost}(k,j,j) = C_b(k-j) + C_j(j) + J(A_{k,j}, A_j) \]  

The cost consists of three parts: the cost of the best plan to get results of \((k-j)\)-time self-join expressed as \(C_b(k-j)\), the cost of the best plan to get results of \(j\)-time self-join expressed as \(C_j(j)\), and the cost to join \(A_{k,j}\) and \(A_j\) expressed as \(J(A_{k,j}, A_j)\). Because the best plan is the one with minimal cost among all these options, it can be calculated by the following formula.

\[ C_b(k) = \text{Min} \left( \text{Cost}(k-1, 1), \text{Cost}(k-2, 2), \ldots, \text{Cost} \left( \left\lfloor \frac{n}{2} \right\rfloor, \left\lfloor \frac{n}{2} \right\rfloor \right) \right) \]

\[ C_b(1) = |A| \]

Using this recursive function, we can calculate the minimal cost to execute a query. By recording the choice in each step, we can find the best plan corresponding to the minimal cost. We prove in the following that this algorithm always output deep-left tree as the best plan.
**Theorem I**: Left-deep tree is the best plan for k-times self-join queries for all k > 0.

We prove this theorem by induction.

**Base**: If k = 1, 2 the only possible plan is left-deep tree. Therefore left-deep tree is the best plan.

**Assumption**: Assume this theorem is true for 0 < k < n. It means that the best plan for A<sub>k</sub> (0 < k < n) is left-deep tree.

**Inductive Step**: Given the above hypothesis, we prove the theorem is true for k=n.

As shown in formula 5.4, C<sub>b</sub>(n) = Min(Cost(n-1, 1), Cost(n-2, 2), ..., Cost(\lceil n/2 \rceil, \lceil n/2 \rceil)), the best plan for n-time self-join is to choose the one with the smallest cost among all these options. We prove Cost(n-1, 1) is smaller than Cost(n-j, j) for 1 < j < \lceil n/2 \rceil) as follows:

\[
\text{Cost}(n, 1) = C_b(n-1) + C_b(1) + J(A_{n,1}, A) \quad \text{(I.a)}
\]

Based on the hypothesis, the best plan to produce A<sub>n-1</sub> is left-deep tree. Therefore,

\[
C_b(n-1) = C_b(n-2) + C_b(1) + J(A_{n-2,1}, A) \quad \text{(I.b)}
\]

We can rewrite formula (I.a) by replacing C<sub>b</sub>(n-1) with the expression in the right side of formula (I.b)

\[
\text{Cost}(n-1, 1) = C_b(n-2) + 2*|A| + |A| * |A| \quad \text{(I.c)}
\]

Because A<sub>n-1</sub> can be calculated by joining A<sub>n-2</sub> and A, the size of A<sub>n-1</sub> can be represented as follows.

\[
|A_{n,1}| = |A_{n-2,1}| * |A| * S(n-2, 1)
\]

Here S(i, j) represents the selectivity of the join operator between A<sub>i</sub> and A<sub>j</sub>. It is defined as

\[
S(i, j) = A_{i,j} / (|A_i| * |A_j|)
\]

Use this formula, we get

\[
\text{Cost}(n-1, 1) = C_b(n-2) + 2*|A| + |A| * |A| + |A| * |A| * S(n-2, 1) \quad \text{(I.c)}
\]

Now, we consider another possible plan to calculate A<sub>n-1</sub> by joining A<sub>n-2</sub> with A 2.

\[
\text{Cost}(n-2, 2) = C_b(n-2) + C_b(2) + J(A_{n-2,2}, A_2)
\]

\[
= C_b(n-2) + |A_2| + |A_{n-2,2}| * |A_2| + J(A_{n-2,2}, A_2)
\]

\[
= C_b(n-2) + |A_2| + |A_{n-2,2}| * |A_2| + J(A_{n-2,2}, A_2) \quad \text{(I.d)}
\]

From formula (I.c) and (I.d), we can conclude that Cost(n-1, 1) < Cost(n-2, 2) holds if and only if the following two expressions hold.

\[
2*|A| < |A_2| \quad \text{(I.e)}
\]

\[
|A| * |A_{n-2,2}| + |A_2| * |A_{n-2,2}| * S(n-2, 1) < |A_{n-2,2}| * |A_2| * S(n-1, 1) \quad \text{(I.f)}
\]

Formula (I.e) is true if |A| > 2. We prove formula (I.f) is true as follows:

\[
|A| * |A_{n-2,2}| + |A_2| * |A_{n-2,2}| * S(n-2, 1) < |A_2| * |A_{n-2,2}| * S(n-1, 1) \quad \text{(I.f)}
\]

According to theorem II, formula (I.g) is true. Therefore, formula (I.f) is true.

Because both formula (I.e) and (I.f) is true, we prove Cost(n-1, 1) < Cost(n-2, 2). In this similar way, we can prove the Cost(n-i, i) < Cost(n-i-1, i+1) for i=2 .. \lceil n/2 \rceil. Therefore, we prove that Cost(n-1, 1) is smallest one among all possibilities. It means joining A<sub>n-1</sub> with A is the best way to calculate A<sub>n</sub>. Based on the hypothesis (the best way to calculate A<sub>n-1</sub> is the left-tree), we can conclude that the best plan to calculate A<sub>n</sub> is a left-deep tree.

**Theorem II**: For a k-time self-join of a relation R, |A_i| / |A_{i-1}| < |A_i| / |A_{i-1}| for 1 < i < k+1

**Proof**: For convenience, we call an intermediate result after x self-join as an x intermediate result. Because the result size for a query is independent of the execution plan for this query. We assume the plan is left-deep tree. Therefore, each x-intermediate result is calculated by joining an (x-1) intermediate result with a tuple from relation R. We assume the logic operator in a sum constraint is >’ or ≥’. According to formula 3.6 in section 3.2, the query condition used to create an x-intermediate result from (x-1) intermediate result is as follows.

\[
\sum_{j=1}^{x} D_j \geq C - \sum_{i=x+1}^{k} \text{Max}(D_i)
\]

It can rewritten as

\[
\sum_{j=1}^{x-1} D_j + D_x \geq C - \sum_{i=x+1}^{k} \text{Max}(D_i) \quad \text{(II.a)}
\]

The first sub-expression on the left side is the sum of attribute D in an (x-1) intermediate result. To create an x intermediate result, we combine it up with value of attribute D in a tuple in relation R and check the constraint. Therefore, the size increase of intermediate results from x-1 to x depends on the number of tuples that combine with an (x-1) intermediate result and satisfy the constraint. We can rewrite previous formula as follows.
\[ D_x \geq C - \sum_{i = x + 1}^{k} \text{Max}(D_i) - \sum_{j = 1}^{x - 1} D_j \]  

(II.b)

It means that only tuples with value of attribute D more than the right side of the expression will combine with the \((x-1)\) intermediate result to create an \(x\) intermediate result. We represent the right side of the expression as a function of \(x\) as follows.

\[ \text{RE}(x) = C - \sum_{i = x + 1}^{k} \text{Max}(D_i) - \sum_{j = 1}^{x - 1} D_j \]

We prove that \(\text{RE}(x)\) gets bigger with the increase of \(x\). For any integer \(t\) and \((t+1)\),

\[ \text{RE}(t+1) - \text{RE}(t) = \left( C - \sum_{i = t + 1}^{k} \text{Max}(D_i) - \sum_{j = 1}^{t - 1} D_j \right) - \left( C - \sum_{i = t + 2}^{k} \text{Max}(D_i) - \sum_{j = 1}^{t - 1} D_j \right) \]

\[ = \text{Max}(D_t) - D_t \]

\[ \geq 0 \]

Because value of the expression on the right side of constraint II.b increases with \(x\), the constraint becomes stricter. Less tuples will be combined with each intermediate result with increase of \(x\). Therefore, for any \(1 < j < i < k+1\), the logic expression \(|A_j|/|A_{i-1}| < |A_i|/|A_{j-1}|\) holds.

In this section, we prove that left-deep tree is always the best plan for \(k\)-time self-join queries.

6 Performance Evaluation

We evaluate the efficiency of the query rewriting technique and the query algorithm for \(K\) results using a benchmark.

6.1 Benchmark database

We use a food database published by USDA [18]. This database contains nutritional facts of more than 7000 types of foods. We build relation fruits, vegetables, beverage, and meats based on the type of food from the food database. Relation Fruits contain 273 types of fruits, 717 types of vegetables, 199 types of beverages, and 1602 types of meats.

We build the database and run all queries on a PostgreSQL 8.0.3 database system. The system is running on a linux box with 4 Intel Xeon CPU 3GHz processors, and 2G bytes memory size.

6.2 Evaluating query rewriting technique

We evaluate the efficiency of query rewriting technique by comparing response time of benchmark queries with or without rewriting. We define two benchmark queries in Figure 5. QA contains one constraint requiring the sum of attribute \(\text{cal}\) more than \(\text{Min}_\text{cal}\). QB contains two constraints requiring the value of the sum of attribute \(\text{cal}\) in a range between \(\text{Min}_\text{cal}\) and \(\text{Min}_\text{cal} + 200\).

\[ \text{QA: SELECT} \ M.\text{name}, V.\text{name}, B.\text{name}, F.\text{name} \]

FROM \ Meats AS M, Vegetables AS V, Beverages AS B, Fruits AS F

WHERE \ M.\text{cal} + V.\text{cal} + B.\text{cal} + F.\text{cal} > \text{Min}_\text{cal} LIMIT K

\[ \text{QB: SELECT} \ M.\text{name}, V.\text{name}, B.\text{name}, F.\text{name} \]

FROM \ Meats AS M, Vegetables AS V, Beverages AS B, Fruits AS F

WHERE \ M.\text{cal} + V.\text{cal} + B.\text{cal} + F.\text{cal} > \text{Min}_\text{cal} \]

AND \ M.\text{cal} + V.\text{cal} + B.\text{cal} + F.\text{cal} < \text{Min}_\text{cal} + 200 \]

LIMIT K

Figure 5. Benchmark query QA and QB for rewriting experiment

There are two ways to improve the query performance for \(K\) results: order the tuples in relations and consider tuples with higher (or lower) attribute values first; or reduce the search space by using query rewriting to introduce new query conditions. We compare six different combinations of these two ways to evaluate a query as shown Table 3. For example, plan I does not sort tuples in relations, and does not rewrite the query. And Plan VI sorts tuples in descending order of attribute \(\text{cal}\), and rewrites the query.

<table>
<thead>
<tr>
<th>Order</th>
<th>Random</th>
<th>Ascend</th>
<th>Descend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without rewriting</td>
<td>Plan I</td>
<td>Plan III</td>
<td>Plan V</td>
</tr>
<tr>
<td>With rewriting</td>
<td>Plan II</td>
<td>Plan IV</td>
<td>Plan VI</td>
</tr>
</tbody>
</table>

Table 3 Execution plans

QA is a query containing only one constraint that specifies the minimal requirement on the total calories of meal. In this case, combinations of foods with larger number of calories have better chance to satisfy this constraint. We compare four plans (plan I, II, V, and VI in Table 3) because sorting tuples in ascending (Plan III, IV) order is obvious not a good choice for this query. We vary the value of \(\text{Min}_\text{cal}\) to change the selectivity of this query. Figure 7 shows the experimental results. X axis is the value of \(\text{Min}_\text{cal}\), the Y axis is query response time. Plan II, which rewrites the query, outperforms plan I, and the difference becomes larger with the increase of \(\text{Min}_\text{cal}\). As shown in Table 1, the average calorie of each serve of food is around 100. With \(\text{Min}_\text{cal}\) more than 700 and increasing, there will be fewer results for the query. Therefore, plan I needs to check more combinations of foods to find given number of meals satisfying the requirements. It displays as the increase of response time. In comparison, the response time of the rewritten query does not increase much. It shows rewriting technique
improves database’s capabilities to find results for a query with low selectivity.

Plan V, which sorts tuples in descending order of attribute cal, outperforms plan II consistently. It means that ranking based method outperforms query-rewriting technique. Also, plan VI, which use both rewriting and sorting, has similar performance as Plan V. It means that rewriting technique does not help ranking based plan to get better performance.

**Figure 6. Comparison of four plans for QA with K=20**

We use QB to evaluate the performance of techniques for more complex queries. Unlike QA, QB contains two ‘conflicting’ constraints that specify a close range of the sum of attribute cal. These two constraints ‘conflict’ because the first constraint favors tuples with higher calories, and the second constraint favors tuples with lower calories. Because we cannot tell intuitively which ranking method is better for this query, we compare six execution plans for this query and show the result in Figure 7.

Two plans Plan III and plan V, which rank tuples in relations based attribute cal and does not use rewriting, shows much higher response time than others. Plan III will check foods with higher calories first. Because QB asks for foods with total calories in a range. Plan III will check foods with too much calories at the beginning of search process for QA, which causes the longer response time. With the increase the minimal requirements on the calories, checking combination with higher calories becomes better as shown in the decrease of the response time with the increase of Min_cal. The Plan V shows a reverse tendency as plan III. This experimental result shows that ranking tuple based on an attribute does not always help if the query contains ‘conflicting’ constraints. Plan IV and VI, which do both tuple ranking and query rewriting, shows similar tendency as plan II, but with slightly worse response time. It shows that ranking actually hurts the query performance of QB.

**Figure 7. Comparing six plans for QB with K=20**

From the experimental results of two benchmark queries, we can see that rewriting technique improves the query performance consistently for different types of sum constraint queries. Sorting based plans, instead, show only good performance for given type of queries (QA in our case).

Also, by comparing ranking only plan (plan III, Plan V) with plans using both rewriting and ranking (plan IV, plan VI), we can see that rewriting never hurts the performance of ranking based plan, and in the case of QB, it remarkably improves the performance of ranking-only plans. So, we can conclude that rewriting technique can be efficiently used with sorting-based plans.

### 6.3 Evaluating sorting technique

In section 4 and 6.2, we show ranking based on one attribute has limitation when the query contains multiple ‘conflicting’ constraints. In this section, we compare our ranking technique, which takes into account all attributes in multiple constraints, with traditional method, which rank tuples based on one constraint.

**Table 4 Plans for the benchmark query**

<table>
<thead>
<tr>
<th>Plans</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan I</td>
<td>Sort tuples in descending order of attribute cal</td>
</tr>
<tr>
<td>Plan II</td>
<td>Sorting tuples in descending order of Vc</td>
</tr>
<tr>
<td>Plan III</td>
<td>Sorting tuples in descending order of Vb6</td>
</tr>
<tr>
<td>Plan IV</td>
<td>Sorting tuples in ascending order of fat</td>
</tr>
<tr>
<td>Plan V</td>
<td>Sorting tuples in ascending order of chol</td>
</tr>
<tr>
<td>Plan N</td>
<td>Sorting tuples based new sorting technique</td>
</tr>
</tbody>
</table>

We use the query in Figure 1 as the benchmark query. This query asks for K combinations of meat, vegetable, fruit, and beverage with required nutrition facts.

```
SELECT  M.name, V.name, B.name, F.name
FROM      Meats AS M, Vegetables AS V,
```
Beverages AS B, Fruits AS F
WHERE M.cal + V.cal + B.cal + F.cal > 900
AND M.cal + V.cal + B.cal + F.cal < 1100
AND M.Vb6 + V.Vb6 + B.Vb6 + F.Vb6 > 2
AND M.Vc + V.Vc + B.Vc + F.Vc > 38
AND M.fat + V.fat + B.fat + F.fat < 9
AND M.chol + V.chol + B.chol + F.chol < 150
LIMIT K

**Figure 8. A benchmark query with multiple sum constraints**

We rewrite this query before executing it, and compare the response time of execution plans shown in Table 4. Plan I to plan VI sort tuples based on values of one attribute as traditional sorting method does. For example, plan I sorts tuple based on the value of attribute cal in descending order because the first condition of the benchmark query asks for the sum of attribute cal more than 900. Plan N use our algorithm proposed in section 5 to rank tuples.

**Figure 9. Response time of different ranking plans**

Figure 9 shows the response time for all these plans. Plan N outperforms all other plans.

**7 Summary**

In this report, we study a special type of queries called sum constraint queries that ask for multiple tuples with the sum of their attributes is more (or less) than a given value. This type of queries appears in document retrieval, multimedia data retrieval, Internet-computing, supply chain management areas for locating multiple tuples with particular aggregation properties. To run this kind of queries efficiently on traditional database engine, we propose a query rewriting technique to infer from the original query conditions new constraints that can be utilized by database search engine to reduce the number of intermediate results and number of tuple read operations. The experimental results show that this technique can improve the query performance remarkably when the number of query results is small.

We also study the execution plan for a special case of sum constraint query that requires joining a relation multiple times. We evaluate all possible plans based on the number of I/O operations. The results show that left-deep tree is the best plan.

**Reference**


