Evaluation of Set-based Queries with Aggregation Constraints

Quoc Trung Tran, Chee-Yong Chan, and Guoping Wang
Department of Computer Science, School of Computing
National University of Singapore
{tqtrung, chancy, wangguoping}@comp.nus.edu.sg

ABSTRACT

Many applications often require finding a set of items of interest with respect to some aggregation constraints. For example, a tourist might want to find a set of places of interest to visit in a city such that the total expected duration is no more than six hours and the total cost is minimized. We refer to such queries as SAC queries for "set-based with aggregation constraints" queries. The usefulness of SAC queries is evidenced by the many variations of SAC queries that have been studied which differ in the number and types of constraints supported. In this paper, we make two contributions to SAC query evaluation. We first establish the hardness of evaluating SAC queries with multiple count constraints and presented a novel, pseudo-polynomial time algorithm for evaluating a non-trivial fragment of SAC queries with multiple sum constraints and at most one of either count, group-by, or content constraint. We also propose a heuristic approach for evaluating general SAC queries. The effectiveness of our proposed solutions is demonstrated by an experimental performance study.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications

General Terms

Algorithms

Keywords

Set-based query, aggregation constraints

1. INTRODUCTION

Many database applications often require finding a set of items of interest with respect to some aggregation constraints. As an example, consider the following database relation which stores information about places of interest: $POI(\underline{name}, price, hour, type, city)$. Here, *name* refers to a place of interest located in *city* with a ticket fee of *price* dollars, *hour* refers to the recommended duration to spend at that place (in hours), and *type* refers to the category of the tourist place (e.g., museum, park). Suppose that Alice plans to find a tour package of places to visit from *POI* that satisfies the following requirements: (R1) all places are located in *NY* city, (R2) the total price is between $300 and $400, (R3) the expected duration is no greater than 10 hours, and (R4) the number of distinct types of places is maximized (to maximize diversity). Observe that Alice's request involves a simple filtering constraint, R1, and three aggregation constraints: two sum constraints (R2 and R3) and a count constraint, R4, to be maximized. The query's result consists of a collection of matching packages where each package is a set of places of interest that satisfy the four requirements.

Besides the "global" aggregation constraints illustrated by the above example, another very useful type of aggregation constraints are "local" *group-by aggregation constraints* that specify an aggregation constraint on specific subgroup(s) or all subgroups of the qualified sets, where a subgroup is defined with respect to an attribute list and optionally specific attribute values. As an example of a group-by count constraint on a specific group, Alice could require that there must not be more than two museum places in a matching package. In contrast, an example of a group-by sum constraint that is specified on each group is when Alice requires that the total visit duration to places of the same type does not exceed four hours.

Yet another useful type of set-based constraints are *content constraints* to specify the presence of certain attribute values (or tuples) in each qualified set. As an example, Alice could specify that each matching package must contain a visit to "Central Park".

We refer to such set-based queries with any combination of aggregation, group-by, and content constraints as `SAC` *queries*. The usefulness of `SAC` queries is evidenced by the variants of `SAC` queries (with different restrictions on the number and types of constraints permitted) that have been studied: OPAC queries for business optimization problems [4], student course planning in the CourseRank project [2], and item recommendations in shopping applications [1].

While `SAC` queries have many useful applications, the queries may not be easily expressed using SQL due to the fact that the qualified sets generally have different cardinalities and the maximum cardinality of the sets is data-dependent. More importantly, even when such complex queries can be

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

expressed and evaluated by SQL engines, their performance can be very poor as demonstrated by our experimental results. Indeed, the evaluation of general SAC queries involving any combination of aggregation, group-by aggregation, and content constraints is an NP-complete problem. Although a number of polynomial [2] and pseudo-polynomial time [4, 5] algorithms have been developed for evaluating simpler fragments of SAC queries, most of the proposed evaluation techniques are heuristic methods for more general SAC query fragments [1].

Contributions. We make two key contributions to the study of SAC query evaluation by addressing two open issues. First, there is the question of whether there exists other non-trivial fragments of SAC queries that can be evaluated in polynomial/pseudo-polynomial time. Currently, there are two fragments of SAC queries that are known to be amenable to efficient evaluation. First, SAC queries with one count constraint and at most one group-by count constraint, where the attribute(s) in the count and group-by constraint are the same, can be evaluated in polynomial time [2]. Second, SAC queries with any number of sum constraints and at most one group-by sum constraint can be evaluated in pseudo-polynomial time [5]. An open question that arises from this fact is: what is the complexity of evaluating SAC queries with more than one count constraint or with a combination of sum and count constraints? In this paper, we address this issue by first establishing that even for SAC queries with only two count constraints, the evaluation problem is already an NP-complete problem *in the strong sense*. Given this negative result for multiple count constraints, we next show that for the SAC query fragment with any number of sum constraints and at most one of either count, group-by, or content constraint, the evaluation problem has pseudo-polynomial time complexity, and we present a dynamic programming approach to evaluate this non-trivial SAC query fragment. Our second contribution is the proposal of a heuristic for evaluating general SAC queries with any combination of sum, count, group-by, and content constraints. The heuristic tries to find a solution that satisfies all the specified constraints but may return an approximate solution that meets only some of the constraints.

2. SAC QUERIES

A **S**et-based query with **A**ggregation **C**onstraints (or SAC query for short) can be expressed in terms of two components, $Q = (Q_{base}, C_{ans})$. Q_{base}, which is the base query of Q, is a conventional relational query that retrieves a set S_{base} of tuples, which serves as the base data for the SAC query. C_{ans} is a set of aggregation/group-by/content constraints such that each subset $S_{ans} \subseteq S_{base}$ that satisfies all the constraints in C_{ans} is a result of the SAC query Q.

The basic aggregation constraint in C_{ans} is a numeric constraint of the form "X *op* c", where X is some expression (to be described), *op* is one of the standard comparison operators ($=, \leq, \geq, <, >$), and c is a non-negative integer constant. The constraints permitted in C_{ans} are of the following five types:

(1) A **sum constraint** is a constraint on the sum of some attribute A_i in S_{ans}, denoted by $sum(A_i)$ *op* c.

(2) A **count constraint** is a constraint on the number of *distinct values* of a subset A' of the attributes in S_{ans}, denoted by $count(A')$ *op* c. When A' is a candidate key attribute of S_{ans}, the count constraint becomes a *cardinality constraint*.

(3) An **optimization constraint** is specified to maximize/minimize an aggregated value, and there are two forms of optimization depending on whether the aggregated value is bounded. A *bounded optimization constraint* is of the form "*opt* $(agg(X))$ *op* c", and an *unbounded optimization constraint* is of the form "*opt* $(agg(X))$". Here, *opt* is either *minimize* or *maximize*; *agg* is an aggregation operator (sum, count); X is either a single attribute (if *agg* is sum or count) or a sequence of attributes (if *agg* is count); and *op* is an non-equality comparison operator. As an example, the constraint $maximize(sum(A_i)) \leq c$ is a bounded optimization constraint on $sum(A_i)$, while the constraint $maximize(sum(A_i))$ is an unbounded optimization constraint on $sum(A_i)$.

(4) A **group-by constraint** is of the form $groupby(A', agg, B')$ *op* c, where A' and B' are subsets of attributes in S_{ans}, and *agg* is an aggregation operator (i.e., sum or count). The constraint requires that if S_{ans} is partitioned into groups of tuples having the same values for attribute(s) A', then each group must satisfy the aggregation constraint $agg(B')$ *op* c. In addition to this basic form of group-by constraint where the same count/sum constraint (i.e. c) is applied to each group, it is also possible to specify individual count/sum constraint for each group by explicitly listing the desired values of c's using the form: $groupby(A' = v_i, agg, B')$ *op* c_i.

(5) A **content constraint** is of the form $contain(A', V)$, where A' is a subset of attributes in S_{ans}, and V is a set of tuple values (of the same arity as the number of attributes in A'). The constraint requires that the projection of S_{ans} on A' must contain the set of tuples V.

In this paper, the SAC query fragment that we study allows C_{ans} to contain at most one group-by constraint. For other types of constraints, C_{ans} can contain any number of them[1]. Furthermore, the domain of the attribute involved in a sum or optimization constraint must contain non-negative values. For simplicity and without loss of generality, cardinality constraints are treated as a form of sum constraints (i.e., summing on a virtual attribute with a value of 1 for each tuple); therefore, we will not explicitly discuss cardinality constraints in this paper.

Since the problem of evaluating a SAC query is very hard in general, we focus on finding some answer for a given SAC query and leave the problems of ranking and/or finding top-k answers for SAC queries as part of our future work. We will use the tour package database illustrated in Figure 1 as our running example.

EXAMPLE 1. *Our example* SAC *query in the introduction can be expressed as follows: The base query Q_{base} corresponds to the SQL query: SELECT * FROM POI WHERE city = "NY", and the aggregation constraints are specified as follows: (R2) $sum(price) \geq 300$, $sum(price) \leq 400$; (R3) $sum(duration) \leq 10$; and (R4) $maximize(count(type))$. The additional local group-by count, local group-by sum, and content constraints discussed are specified, respectively, as follows: $groupby(type = museum, count, *) \leq 2$, $groupby(type, sum, duration) \leq 4$, and $contain(name, "Central\ Park")$.* □

[1]Note that when there are multiple optimization constraints, the query becomes a skyline query.

	name	price	hour	type	city
t_1	L1	50	2	museum	X
t_2	L2	70	3	park	X
t_3	L3	40	1	theatre	X
t_4	L4	25	2	museum	Y
t_5	L5	90	4	museum	Y
t_6	L6	20	1	shopping	Z

Figure 1: Running Example: Place of Interest (POI) relation

		city	hour	type
P_1	t_1	X	2	museum
	t_2	X	3	park
	t_3	X	1	theatre
P_2	t_4	Y	2	museum
	t_5	Y	4	museum
P_3	t_6	Z	1	shopping

(a) S_{base}

$SV_1 = \{1, 2, 3, 4, 5, 6\}$
$SV_2 = \{2, 4, 6\}$
$SV_3 = \{1\}$

(b) SV-sets

Figure 2: Partition POI using $city$ attribute

3. MULTI-SUM SAC QUERIES

With the exception of two fragments of SAC queries, namely, SAC queries with one count constraint and at most one group-by count constraint [2] and SAC queries with any number of sum constraints and at most one group-by sum constraint [5], most of the proposed techniques for evaluating more general SAC queries are heuristic solutions [1]. This raises the interesting open question of what is the complexity of evaluating SAC queries with more than one count constraint or SAC queries with a combination of sum and count constraints.

To address this issue, we first establish the following result on the evaluation of SAC queries with multiple count constraints; the proof is given elsewhere [6].

THEOREM 1. *Consider a* SAC *query Q where C_{ans} contains exactly two count constraints on two distinct attributes. The problem of evaluating Q is NP-complete.* □

Given the above negative result for SAC queries with multiple count constraints, we next ask the question of whether SAC queries with at most one count constraint can be efficiently evaluated. In this section, we show that for the SAC query fragment with any number of sum constraints and at most one of either count, group-by, or content constraint, the evaluation problem has pseudo-polynomial time complexity, and we present a dynamic programming approach to evaluate this non-trivial SAC query fragment. Our dynamic programming approach can also evaluate SAC query that includes any number of sum constraints and one count, one group-by, and one content constraint, where the attribute(s) used in these constraints are the same.

3.1 DP: Conceptual Approach

In this section, we present a novel, pseudo-polynomial time algorithm, termed DP, for evaluating SAC queries with any number of sum constraints and at most one of either count, group-by, or content constraint. For convenience, we refer to this SAC query fragment as *multi-sum* SAC *queries*.

To simplify the presentation, we first explain how DP evaluates a simpler subset of multi-sum SAC queries with multiple sum and a single count constraints, and then briefly discuss the generalization of DP to solve the general multi-sum SAC queries in Section 3.3.

For simplicity and without loss of generality, we explain the evaluation for a SAC query $Q = (Q_{base}, C_{ans})$ where C_{ans} contains the following two constraints on two attributes A and B and the domain of A contains integer values[2].

- Sum maximization constraint: $maximize(sum(A)) \leq K$, and
- Count constraint: $count(B) = m$.

[2]If the domain of A contains real values, DP approximates the real values by integer values.

Let ℓ denote the number of distinct B attribute values in S_{base}. Recall that S_{base} is the set of base data retrieved by the base query Q_{base}. DP is based on the following two-phase dynamic programming approach.

First, DP partitions S_{base} using the B attribute into ℓ partitions $P_1, \cdots, P_\ell$. For each partition P_i, DP derives $SV_i = \{V \in [1, K] \mid \exists\ P_{s,i} \subseteq P_i\ s.t. \sum_{t \in P_{s,i}}(t.A) = V\}$. We say that a value $V \in SV_i$ is the *representation* of a set $P_{s,i} \subseteq P_i$ iff $\sum_{t \in P_{s,i}}(t.A) = V$. Note that in general, V can represent more than one set $P_{s,i}$; however, DP only records one set $P_{s,i}$ that V represents. This task is a subset-sum problem that can be solved using dynamic programming.

In the next step, DP selects m SV_i sets and one value from each selected set SV_i so that the summation of these selected values is maximized and does not exceed K. This task can be solved using dynamic programming. Without loss of generality, assume that the selected sets in this step are $SV_1, \cdots, SV_m$; and the corresponding value selected in each SV_i is v_i, $i \in [1, m]$. Since each v_i represents a subset $P_{s,i} \subseteq P_i$, DP unions these $P_{s,i}$, $i \in [1, m]$, as the final answer S_{ans}. Clearly, S_{ans} has $count(B) = m$, as each $P_{s,i}$ "contributes" a distinct B value. Furthermore, S_{ans} has $sum(A)$ maximized without exceeding K because $\sum_1^m(v_i)$ is maximized and does not exceed K.

EXAMPLE 2. *Assume a user wants to find a package of places to visit from POI with the following two constraints: (1) $maximize(sum(hour)) \leq 7$, and (2) $count(city) = 2$.* DP *first partitions S_{base} into three partitions, P_1 to P_3, using city attribute, as shown in Figure 2(a).* DP *then derives the corresponding three sets, SV_1 to SV_3, shown in Figure 2(b). For instance, $v_{1,5} = 5 \in SV_1$, since $v_{1,5}$ represents for the subset $P_{1,s} = \{t_1, t_2\}$; i.e., $\sum_{t \in P_{1,s}}(t.hour) = 5$.*

In the next step, DP *selects SV_1 and SV_2; and chooses $5 \in SV_1$ and $2 \in SV_2$ where their summation is maximized and does not exceed 7. Since 5 represents for the set $P_{1,s} = \{t_1, t_2\}$ and 2 represents for the set $P_{2,s} = \{t_4\}$,* DP *unions $P_{1,s}$ and $P_{2,s}$ to return $S_{ans} = \{t_1, t_2, t_4\}$. Note that S_{ans} is not the only solution; another possible solution is $\{t_4, t_5, t_6\}$.* □

3.2 DP: Algorithm

We now elaborate on the details of DP for evaluating a SAC query involving two attributes A and B as given in Section 3.1. For simplicity and without loss of general-

ity, let the domain of the B attribute values in S_{base} be $dom(B) = \{1, 2, \cdots, \ell\}$[3].

For each $b \in dom(B)$, let S^b_{base} and $S^{\leq b}_{base}$ be a subset of S_{base} defined in Equations 1 and 2, respectively. Let $E[1 \cdots \ell, 1 \cdots K]$ be a two-dimensional matrix, where each cell $E[b, V]$ is a boolean value defined in Equation 3. Each row $E[b, .]$ is a subset-sum problem that can be solved with a time complexity of $O(K|S^b_{base}|)$. Therefore, the entire matrix E can be constructed in $O(K|S_{base}|)$.

Let $\mathcal{D}[1 \cdots \ell, 1 \cdots m, 1 \cdots K]$ be a three-dimensional matrix, where each cell $\mathcal{D}[b, d, V]$ is a boolean value defined in Equation 4.

$$S^b_{base} = \{t \in S_{base} \mid t.B = b\} \quad (1)$$

$$S^{\leq b}_{base} = \{t \in S_{base} \mid t.B \leq b\} \quad (2)$$

$$E[b, V] = true \text{ iff } \exists\ S \subseteq S^b_{base} \text{ s.t. } \sum_{t \in S}(t.A) = V \quad (3)$$

$$\mathcal{D}[b, d, V] = true \text{ iff} \exists\ S \subseteq S^{\leq b}_{base} \text{ s.t. } |\pi_B(S)| = d \wedge \sum_{t \in S}(t.A) = V \quad (4)$$

DP can find a solution if there exists a maximum value $V_{max} \leq K$ such that (1) $\mathcal{D}[\ell, m, V_{max}] = true$, and (2) for every value $V > V_{max}$, $\mathcal{D}[\ell, m, V] = false$. We have the following recurrence relation:

$$\mathcal{D}[b, d, V] = \mathcal{D}[b-1, d, V] \ \vee$$
$$\exists\ V' \in [1, V]\ s.t.\ (E[b, V'] = 1\ \wedge\ \mathcal{D}[b-1, d-1, V-V'] = 1)$$

The recurrence relation indicates that $\mathcal{D}[b, d, V]$ can be derived from either (1) $\mathcal{D}[b-1, d, V]$ if we do not select any tuples from S^b_{base}, or (2) $\mathcal{D}[b-1, d-1, V-V']$ if we select a subset of tuples S' from S^b_{base} with $\sum_{t \in S'}(t.A) = V'$.

The computation of each $\mathcal{D}[b, d, V]$ requires at most V look up operations on the corresponding row $E[b, .]$ in the E matrix. Thus, the time to build matrix $\mathcal{D}$ in the worst case is $O(m\ell \sum_{V=1}^{K}(V)) = O(K^2 m\ell)$.

Deriving S_{ans}. In addition to the main matrix $\mathcal{D}$, DP uses another matrix $DTrace[\ell, m, K]$ that has the same dimensions as $\mathcal{D}$ to derive S_{ans}. Each cell $DTrace[b, d, V]$ is set to either (1) a value 0 if $\mathcal{D}[b, d, V]$ is derived from $\mathcal{D}[b-1, d, V]$, or (2) a value $V' > 0$ if $\mathcal{D}[b, d, V]$ is derived from $\mathcal{D}[b-1, d-1, V-V']$ and $E[b, V']$.

To derive a set of returned tuples S_{ans}, DP first determines the maximum value $V_{max} \leq K$ such that $\mathcal{D}[\ell, m, V_{max}] = true$. There are two cases to consider:

- If $DTrace[\ell, m, V_{max}] = 0$, then S_{ans} is the set of tuples that makes $\mathcal{D}[\ell - 1, m, V_{max}] = true$.
- Otherwise, if $DTrace[\ell, m, V_{max}] = V'$, then S_{ans} is the union of the set of tuples that makes $D[\ell - 1, m - 1, V_{max} - V'] = true$ and the set of tuples that makes $E[\ell, V'] = true$.

The technique to derive a set of tuples that makes $E[\ell, V'] = true$ follows a standard procedure for solving the subset-sum problem. We briefly describe this procedure in the following.

[3]In general, we can easily map an arbitrary set of ℓ values into the set $\{1, 2, \cdots, \ell\}$.

Assume that $S^\ell_{base} = \{t_1, \cdots, t_y\}$. To compute $E[\ell, .]$, DP builds a two-dimensional matrix $F[1 \cdots y, 1 \cdots K]$ with the following recurrence equation:

$$F[i, V] = F[i-1, V]\ \vee\ F[i-1, V - t_i.A]$$

DP maintains another matrix, denoted as $FTrace[1 \cdots y, 1 \cdots K]$, that has the same dimensionality as F. Each $FTrace[i, V]$ keeps track of how $F[i, V]$ is derived; i.e., $FTrace[i, V]$ is set to either (1) $false$ if $F[i, V]$ is derived from $F[i-1, V]$; or (2) $true$, otherwise.

To find a subset S^ℓ of tuples that make $E[\ell, V] = true$, DP traces from $FTrace[y, V]$. If $F[y, V] = false$, then S^ℓ is the set of tuples that makes $F[y-1, V] = true$. Otherwise, if $F[y, V] = true$, then S^ℓ is the union of $\{t_y\}$ and the set of tuples that makes $F[y-1, V - t_y.A] = true$.

Complexity. The space complexity of DP is $O(K|S_{base}| + Km\ell)$ to keep the matrices for the recurrence relations in the main memory. The time complexity of DP is the summation of the following three components: (1) partitioning S_{base} based on B attribute (denoted by T_{part}), (2) computing E matrix, and (3) computing $\mathcal{D}$ matrix. To partition S_{base} based on the B attribute, `Greedy` augments the base query with an *ORDER BY* clause on B and evaluates this derived query to obtain the set of partitions of S_{base}. Thus, T_{part} is the time to execute the derived query on the DBMS. The time complexity of DP is, therefore, $O(T_{part} + K|S_{base}| + K^2 m\ell)$.

Approximation version of DP. When K and/or ℓ is large, the space required by DP might exceed the available main memory. In these cases, DP needs to reduce the space requirement by scaling down the domain values of the attribute used with the sum constraint (i.e., A attribute) by some factor $\mathtt{c_f}$; thus, K will be replaced by $K/\mathtt{c_f}$. The solution of DP is approximate in these cases.

3.3 DP: Generalization

The DP approach can be generalized to evaluate multi-sum SAC queries when C_{ans} includes more than one optimization constraint. The idea is to build a matrix for the dynamic programming technique similar to $\mathcal{D}$ and find all the "skyline" values in this matrix. In particular, assume C_{ans} includes $(n+1)$ constraints: $maximize(sum(A_i)) \leq K_i$ for $i \in [1, n]$, and $count(B) = m$. DP builds a $(n+2)$-dimensional matrix $\mathcal{D}[1 \cdots \ell, 1 \cdots m, 1 \cdots K_1, \cdots, 1 \cdots K_n]$. To derive a result set S_{ans}, DP finds all "skyline" cells $\mathcal{D}[\ell, m, V_1, \cdots, V_n] = 1$ such that there does not exist any tuple of values $(V'_1, \cdots, V'_n)$ where (1) $\mathcal{D}[\ell, m, V'_1, \cdots, V'_n] = 1$, (2) $V'_i \geq V_i$, for all $i \in [1, n]$, and (3) at least one of V'_i is strictly greater than V_i.

Similarly, the DP approach can be generalized to solve multi-sum SAC query when C_{ans} includes any number of sum constraints and at most one of either content or group-by constraint. The idea is also based on a two-phase dynamic programming approach. In the first phase, DP partitions S_{base} using attribute(s) for the content or group-by constraints and derives SV-sets for each partition of S_{base} with respect to the content or group-by constraints. The second phase manipulates the derived SV-sets to compute the answer set. The details are given elsewhere [6].

4. GENERAL SAC QUERIES

In this section, we examine the evaluation of general SAC

queries involving any combination of aggregation, group-by, and content constraints. By Theorem 1, it follows that this evaluation problem is NP-complete. We therefore present a heuristic solution, termed `Greedy`, to evaluate general `SAC` queries. An answer set computed by our heuristic could be suboptimal in the sense that the answer set does not satisfy all the required constraints.

For ease of presentation, our discussion is organized into three cases from the simplest to the most general case.

4.1 Count Constraints

We first discuss the simplest scenario where all the constraints in C_{ans} are count constraints. For simplicity and without loss of generality, we consider a `SAC` query with two count constraints: $count(B_i) = m_i$, $i \in [1, 2]$, where $m_1 \leq m_2$. Our `Greedy` heuristic is based on the following result.

LEMMA 1. *If there exists a subset $S_{count} \subseteq S_{base}$ that has $count(B_1) = m_1$ and $count(B_2) \geq m_2$, then there exists a subset $S_{ans} \subseteq S_{count}$ that has $count(B_1) = m_1$ and $count(B_2) = m_2$.*

Proof of Lemma 1. Given a subset $S_{count} \subseteq S_{base}$ that has $count(B_1) = m_1$ and $count(B_2) \geq m_2$, we first pick m_1 arbitrary tuples in S_{count} that have m_1 distinct B_1 values to put into S_{ans}. The number of distinct B_2's values in S_{ans} is currently not greater than m_1 and therefore is also not greater than m_2, since our assumption is $m_1 \leq m_2$. We then need to insert some tuples from $(S_{count} - S_{ans})$ into S_{ans} to increase the number of distinct B_2's values in S_{ans} into m_2. This task is accomplished by performing $m_2 - |\pi_{B_2}(S_{ans})|$ steps. In each step, we pick a tuple in $(S_{count} - S_{ans})$ to insert into S_{ans} in such a way that the number of distinct B_2's values in the resultant S_{ans} increases by 1. □

Using Lemma 1, `Greedy` will derive a set $S_{count} \subseteq S_{ans}$ that has $count(B_1) = m_1$ and $count(B_2)$ is as *large* as possible. The rationale is that if S_{count} has $count(B_2) \geq m_2$, then we can derive S_{ans} from S_{count} satisfying all the constraints. The details of `Greedy` are as follows.

`Greedy` first partitions S_{base} using the values of B_1 attribute, and performs m_1 iterations to insert m_1 partitions of S_{base} into S_{count}. At each iteration, `Greedy` considers all potential partitions in S_{base}, and chooses the "best" partition to insert into S_{count} such that the resultant S_{count} has the largest number of distinct B_2's values. After m_1 iterations, there are two cases to consider. If $|\pi_{B_2}(S_{count})| \geq m_2$, `Greedy` derives S_{ans} based on Lemma 1 such that S_{ans} satisfies all the count constraints from S_{count}. Otherwise, if $|\pi_{B_2}(S_{count})| < m_2$, `Greedy` returns $S_{ans} = S_{count}$ as an approximate result that does not satisfy the count constraint on B_2.

Complexity. The space complexity of `Greedy` is $O(|S_{base}|)$, since in the worst case, `Greedy` stores all distinct B_2 values in the main memory. The time complexity of `Greedy` is $O(T_{part} + T_{alg})$, where (1) T_{part} is the time to partition S_{base} based on B_1 attribute, and (2) T_{alg} is the time to derive S_{count} and then S_{ans}. Similar to `DP`, `Greedy` also augments the base query with an *ORDER BY* clause on B_1 and evaluates this derived query to obtain the set of partitions of S_{base}. We have $T_{alg} = m_1|S_{base}|$, since `Greedy` performs m_1 iterations and basically scans all tuples in S_{base} in each iteration.

EXAMPLE 3. *Consider a user who wants to find a tour package consisting of places of interest from two different cities and with three different types of activities.* `Greedy` *first derives a set S_{count} that has $count(city) = 2$ and $count(type)$ as large as possible. For this task,* `Greedy` *partitions $S_{base} = POI$ into three partitions using city attribute, as shown in Figure 2(a). In the first iteration,* `Greedy` *selects partition P_1 to put into S_{count}, since P_1 contains the largest number of distinct type values. In the second iteration,* `Greedy` *selects P_3 to put into S_{count}, since P_3 increases the number of distinct type values in S_{count} the most. Note that* `Greedy` *does not select P_2, since it does not help to increase $count(type)$ in S_{count}. Thus, $S_{count} = P_1 \cup P_3 = \{t_1, t_2, t_3, t_6\}$.*

Since S_{count} has $count(type) = 4$, `Greedy` *can derive the exact answer for this case. Using Lemma 1,* `Greedy` *first selects $t_1 \in P_1$ and $t_6 \in P_3$ to put into S_{ans} for $count(city) = 2$ in S_{ans}. Finally,* `Greedy` *puts t_2 into S_{ans}. Thus, $S_{ans} = \{t_1, t_2, t_6\}$.* □

Generalization. The other cases of count constraints involving inequality comparison operator can be reduced to the case of count constraints with equality operator. For instance, if the constraints are $count(B_1) \leq m_1$ and $count(B_2) \geq m_2$, we can use the above solution to find sets of tuples satisfying $count(B_1) = x$ and $count(B_1) = y$ for $0 \leq x \leq m_1$ and $m_2 \leq y \leq |B_2|$, where $|B_2|$ denotes the number of distinct B_2's values in S_{base}.

When there are more than two count constraints, assume these constraints are $count(B_i) = m_i$, $i \in [1, k]$ with $m_k = \max_{i=1}^{k}(m_i)$. As before, `Greedy` partitions S_{base} using B_1, $\cdots$, B_{k-1} attributes. At each iteration, `Greedy` selects one partition from S_{base} to insert into S_{ans} such that: (1) the constraints on B_1, $\cdots$, B_{k-1} can still be satisfied in the sense that $count(B_i) \leq m_i$ for $i \in [1, k-1]$, and (2) the number of distinct B_k values in the resultant S_{ans} is the largest. `Greedy` terminates the iteration when none of partitions satisfies both of these conditions.

4.2 Count & Sum Constraints

In this section, we consider the more complex scenario when there is a combination of count and sum constraints. For simplicity and without loss of generality, we consider a `SAC` query with two constraints: (1) $maximize(sum(A)) \leq K$ and (2) $count(B) = m$.

Our `Greedy` heuristic tries to satisfy the "easier" type of constraints before considering the "harder" constraints. Specifically, `Greedy` considers the constraints in the following order: count, sum, and finally the optimization constraint.

To satisfy the count constraint, `Greedy` can select an arbitrary subset $S_{count} \subseteq S_{base}$ that has $|\pi_B(S_{count})| = m$. Observe that the more tuples that S_{count} has, the more flexibility we have to select a subset of tuples from S_{count} to satisfy other constraints. Thus, `Greedy` finds S_{count} that has the cardinality as *large* as possible among all possible S_{count}'s. For this task, `Greedy` partitions tuples in S_{base} based on their B's values, and chooses m partitions that have the *largest* cardinalities to form S_{count}.

To satisfy the sum constraint, `Greedy` partitions tuples in S_{count} based on their B attribute values, and selects the tuple that has the smallest A value in each partition of S_{count} to insert into S_{ans}. If $\sum_{t \in S_{ans}}(t.A) > K$, it implies that any other subset of S_{count} will not satisfy both the count

and sum constraints; therefore, Greedy returns S_{ans} as an approximate result in this case.

Finally, Greedy handles the optimization constraint by adding some tuples from $(S_{count} - S_{ans})$ into S_{ans}. This task is a subset-sum problem: select a subset of tuples from $(S_{count} - S_{ans})$ that has $max(sum(A)) \leq K - \sum_{t \in S_{ans}}(t.A)$. It is important to note that we cannot add any tuples from $(S_{base} - S_{count})$ into S_{ans} since it would increase the number of distinct B values in S_{ans} and violate the count constraint.

Complexity. The space complexity of Greedy is $O(K|S_{count}|)$ to maintain the matrix for dynamic programming in the last step. The running time of Greedy is $O(T_{part} + T_{SSP})$ where: (1) T_{part} is the time to partition S_{base} using B attribute and select m partitions that have the largest cardinalities, and (2) T_{SSP} is the running time of the solver for the subset-sum problem in the last step of Greedy. For T_{part}, Greedy augments the base query with the following: a *GROUP BY* clause on B, an *ORDER BY* clause on $COUNT(*)$, and a *LIMIT* clause to select the top-m highest cardinality partitions. For T_{SSP}, Greedy uses the conventional pseudo-polynomial algorithm to solve subset-sum problem. Thus, $T_{SSP} = O(K|S_{count}|)$.

EXAMPLE 4. *We reconsider Example 2 and use* Greedy *to solve the problem that finds a package having* $count(city) = 2$ *and* $maximize(sum(hour)) \leq 7$. Greedy *first partitions* S_{base} *using city attribute, and derives* $S_{count} = P_1 \cup P_2$, *which has the largest cardinality from all possible* S_{count}*'s.*

To satisfy the sum constraint, Greedy *selects* $t_3 \in P_1$ *and* $t_4 \in P_2$, *which has the smallest hour value to put into* S_{ans}. *Thus, we have* $\sum_{t \in S_{ans}}(t.hour) = 3$.

For the optimization constraint, Greedy *will select some tuples from* $S_{count} - S_{ans} = \{t_1, t_2, t_5\}$ *that have* $maximize(sum(hour)) \leq 4$ *to put into* S_{ans}. *Thus,* Greedy *will put* t_5 *into* S_{ans}.

In summary, $S_{ans} = \{t_3, t_4, t_5\}$ *and the solution of* Greedy *is turned out to be optimal in this case.* □

Generalization. The described algorithm can be generalized for cases where C_{ans} consists of any number of count and sum constraints, similar to the generalization described in Section 4.1. For instance, consider the situation with count constraints on attributes $A_1, \cdots, A_k$ and sum constraints $sum(B_i) < K_i$, for $i \in [1, \ell]$. Greedy first uses the described algorithm in Section 4.1 to derive S_{count} that satisfies the count constraints. At the end of this step, we can view S_{count} as being partitioned into n partitions using attributes $A_1, \cdots, A_k$. In the next step, Greedy selects one tuple in each partition of S_{count} to put into S_{ans}. The key point is how to select one tuple from each partition so that S_{ans} satisfies the sum constraint. Observe that if every selected tuple t in each partition has $t.A_i < K_i/n$, for $i \in [1, \ell]$, then these selected tuples together will satisfy all the sum constraints. Thus, our Greedy heuristic selects one tuple in each partition that satisfies the *largest* number of conditions "$t.A_i < K_i/n$" to put into S_{ans}.

4.3 General Case

In the general case when C_{ans} includes any combination of constraints, Greedy also follows the "easier-to-harder-constraint" principle so as to allow for more flexibility to satisfy all the constraints. In particular, Greedy considers the constraints in the following order: (1) content constraints, (2) count constraints, (3) sum constraints, and (4) group by together with optimization constraints. The absence of any constraints (e.g., count) allows Greedy to skip the corresponding step (e.g., skip the second step for count constraints). The details are given elsewhere [6].

5. BOUNDED SAC QUERIES

We now switch our attention to a special fragment of SAC queries, referred to as *bounded* SAC *queries*. A bounded SAC query has a bounded cardinality constraint to bound the cardinality of the qualified sets. As a simple example, Alice might be interested only in tour packages containing at most four places to visit. Bounded SAC queries are of interest because they can be processed using two existing approaches. In this section, we briefly describe these techniques, namely DirectSQL and MS, and compare these techniques against our proposed DP and Greedy in Section 6.

Since bounded SAC queries can be expressed directly using SQL, the first alternative evaluation method, denoted by DirectSQL, is to use relational database engines. Consider a SAC query Q with Q_{base} as "SELECT $*$ FROM R" where there is a cardinality constraint in Q that sets the cardinality of qualified sets to a value k. Assuming that the key of relation R is an attribute id, Q can be expressed using SQL as follows:

SELECT $*$
FROM $R\ p_1, \cdots, R\ p_k$
WHERE $p_1.id < p_2.id$ AND $p_2.id < p_3.id$ AND $\cdots$ AND $p_{k-1}.id < p_k.id$ AND qualification

Here, qualification refers to the predicates that represent the remaining constraints in C_{ans}. Thus, each qualified set is returned as a tuple in the query's result. We now explain how to translate the various types of constraints in C_{ans} into SQL predicates.

Content constraint. For each content constraint of the form $content(A, v)$, DirectSQL needs to ensure that at least one instance p_i of R contains a tuple t that has $t.A = v$. The constraint can be expressed by the following SQL predicate: $(p_1.A = v$ or $\cdots$ or $p_k.A = v)$.

Sum constraint. A sum constraint of the form $sum(A) < K$ can be simply translated into the predicate: $(p_1.A + \cdots + p_k.A < K)$.

Count constraint. For a count constraint on an attribute A, we need to count the number of distinct values among $p_1.A$, $\cdots$, $p_k.A$. This can be achieved by using SQL's *case* construct to map each $p_i.A$ value to either 0 or 1 and comparing the sum of the mapped values against the desired count value. Specifically, each $p_i.A$ is mapped to 1 if the value of $p_i.A$ is not equal to any of the preceding $p_j.A$ values, $j < i$; otherwise, it is mapped to 0. For example, the count constraint "$count(A) = m$" is translated as follows:

(1 +
(case when $p_2.A \neq p_1.A$ then 1 or 0 end) + $\cdots$ +
(case when $p_k.A \neq p_1.A$ and $\cdots$ *and* $p_k.A \neq p_{k-1}.A$ then 1 or 0 end)) $= m$

Group-by constraint. Group-by constraints are translated similarly as count constraints using the *case* construct. For example, consider the group-by constraint *groupby*(A, *agg*, B). For each $p_i.A$, $i \in [1, k]$, the translation considers other $p_j.A$, $j \neq i$, that has $p_j.A = p_i.A$ and performs the aggregation function *agg* on B attribute w.r.t. the tuples of p_i and p_j.

Table	# Tuples
adult	45222
part	200000
supplier	10000
track	10000000

Table 1: Sizes of Test Data sets (number of tuples)

Optimization constraint. An optimization constraint is translated essentially by sorting the query's result based on the aggregated attribute value and retrieving only the first tuple by using SQL's *ORDER BY* and *LIMIT* constructs.

For bounded SAC queries where the set cardinality is constrained to be a fixed value (e.g., $count(place) = 4$), a second alternative evaluation method is to apply a recently proposed technique MS [8]. The MS technique was actually designed for solving a more general problem, specifically finding preference sets (with multiple optimization constraints) over fixed-cardinality sets of items. However, the technique there can be used to evaluate bounded SAC queries as well. Further comments on MS are given in Section 7.

6. EXPERIMENTAL STUDY

In this section, we study the effectiveness and the efficiency of our proposed techniques to evaluate SAC queries. The algorithms compared include the dynamic programming approach, DP, that evaluates SAC queries with any number of sum constraints and at most one count, content, or group-by constraint; and Greedy, a heuristic approach for evaluating general SAC queries with any combination of constraints. In addition, we also compare DP and Greedy against two other methods for bounded SAC queries: (1) a direct approach using SQL queries, denoted by DirectSQL, and (2) the approach MS proposed in [8].

We used three real data sets for the experiments: Adult[4], TPCH (with a database size of 1GB), and a music data set containing $10,000,000$ songs and $500,000$ artists[5]. We used four test queries on Adult data set (Q_1 to Q_4), two queries (Q_5, Q_6) on TPCH data set, two queries (Q_7, Q_8) on the music data set, and another two queries Q_1' and Q_1'' that differ slightly from Q_1. Among these queries, Q_1 - Q_3 and Q_5 - Q_8 are multi-sum SAC queries, Q_4 is a general SAC query, and Q_1' and Q_1'' are bounded SAC queries. The test data and queries are given in Tables 1 and 2, respectively. The third column in Table 2 shows the number of tuples in the query result of the base query of each SAC query.

We used PostgreSQL 8.3 for our database system, and all algorithms (DP, Greedy, DirectSQL, and MS) were coded using C++ and compiled with GNU C++ compiler. Our experiments were conducted on a dual-core, 2.33GHz PC running Linux with 3.25GB of RAM and a 250GB hard disk.

6.1 Comparison between DP and Greedy

This section compares DP and Greedy to evaluate SAC queries in terms of the quality of computed results and the running time. Both methods are applicable for all test queries except for query Q_4, which is a general SAC query that consists of all kinds of constraints supported in this paper and cannot be handled by DP. We report the running times of these methods to return one result set for each test query.

Quality of Computed Results. We measured the quality of a query result evaluation in terms of two metrics: (1) the number of query constraints that are satisfied by the computed result, and (2) the aggregate value returned for the optimization constraint. For all of our test queries, both DP and Greedy can find solutions that satisfy all the required constraints. Thus, we compare the quality of computed results for these test queries using the *ratio* between the aggregated value returned by DP over the aggregated value returned by Greedy, referred to as *relative* aggregated value of DP over Greedy. Since the optimization constraint in queries Q_1 to Q_8 is maximization, the higher the relative aggregated value of DP over Greedy, the better the solution of DP is in comparison with Greedy. The relative aggregated value of DP over Greedy for queries Q_1 to Q_8 are given in the second column in Table 3.

For queries Q_1 to Q_3, which involve small relations, DP returned optimal solutions while Greedy returned high-quality solutions; i.e., the aggregated values returned by Greedy are around 3% lower than the optimal values by DP.

Query Q_4 is an example of a general SAC query that contains all kinds of constraints supported in this paper and DP cannot handle. Thus, the DP result for Q_4 is not shown in Table 3. For this query, Greedy can find one set of tuples that satisfies all the constraints.

For queries that involve large data sets (Q_5 to Q_8), since the constructed matrices for the dynamic programming are too large to fit in the main memory, DP used its approximation version to scale down the domain values of the attributes used with the aggregated constraints (e.g., length, retail price attributes). For Greedy, with the heuristic strategy, Greedy first selected a set of tuples satisfying the count constraints, thus reducing the number of tuples to be considered by dynamic programming for the sum optimization constraints. Therefore, the solution of Greedy can be better than DP in these cases. In fact, the results of Greedy for Q_5 and Q_6 are slightly better than DP. For queries Q_7 and Q_8, both the number of involved tuples (3727521 tuples) as well as the number of distinct values of the attribute associated with the count constraint (270352 and 2003678 values, respectively) are large. Here, Greedy returns much better quality result than DP for Q_7 and Q_8. The aggregated values returned by Greedy are around 1.5 - 3.5 times larger than the ones by DP and are nearly equal to the maximum aggregated values required by the queries.

To study how far the aggregated values returned by Greedy and DP are from the optimal solutions, we varied the bounds on the aggregate values of queries Q_1 and Q_8. We report the results on queries Q_3 and Q_7 in Figure 3 (the same trends are observed for other queries). For Q_3, we vary the bounded optimal value from the set $\{500, 1000, 2000, 3000, 40000, 320000\}$. For Q_7, we vary the bounded value from the set $\{6\times10^6, 12\times10^6, 18\times10^6, 24\times10^6, 30\times10^6\}$. The results again show that when the matrices constructed by dynamic programming can fit in the main memory, the optimal value returned by DP is slightly better than Greedy. In contrast, when the matrices cannot be fit in main memory, the solution of Greedy is better than DP. For all cases, the optimal values obtained by Greedy are very close (e.g., more than 95%) to the bounded optimal values, which indicates that Greedy returns high quality solutions.

[4]http://archive.ics.uci.edu/ml/datasets/Adult
[5]http://musicbrainz.org/

Query	Constraints	Size
Q_1	$Q_{base} = \pi_*(adult)$; $maximize(sum(edunum)) \leq 5000$; $count(race) = 2$	45222
Q_2	$Q_{base} = \pi_*(adult)$; $maximize(sum(edunum)) \leq 1000$; $groupby(nativecountry, count, *) \leq 5$	45222
Q_3	$Q_{base} = \pi_*(adult)$; $maximize(sum(capitalloss)) \leq 1000$; $2 \leq count(occupation) \leq 4$;	45222
Q_4	$Q_{base} = \pi_*(adult)$; $maximize(sum(capitalloss)) \leq 2000$; $count(nativecountry) = 2$; $count(race) = 2$ $content(nativecountry$, "United-States"); $groupby(nativecountry, race, count, *) \leq 2$	45222
Q_5	$Q_{base} = \pi_*(part)$; $maximize(sum(retailprice)) \leq 80000$; $count(brand) = 4$	200000
Q_6	$Q_{base} = \pi_*\sigma_{acctbal>0}(supplier)$; $maximize(sum(acctbal)) \leq 150000$; $8 \leq count(nation) \leq 10$	9114
Q_7	$Q_{base} = \pi_*\sigma_{length>240000}(track)$; $maximize(sum(length)) \leq 30000000$; $count(artist) = 5$; content(artist, "Bob Dylan")	3727521
Q_8	$Q_{base} = \pi_*\sigma_{length>240000}(track)$; $maximize(sum(length)) \leq 18000000$; $4 \leq count(title) \leq 6$; content(title, "Jingles Bell")	3727521
Q_1'	$Q_{base} = \pi_*\sigma_{id\leq 60}(adult)$; $maximize(sum(edunum)) \leq 50$; $count(race) = 2$; $count(*) = x$	60
Q_1''	$Q_{base} = \pi_*\sigma_{id\leq 500}(adult)$; $maximize(sum(edunum)) \leq 1000$; $count(race) = 2$; $count(*) = x$	500

Table 2: SAC queries for experiments

Query	Relative aggregated value DP/Greedy	Running time (secs) DP	Running time (secs) Greedy
Q_1	1/1	7.7	7.5
Q_2	1/1	10	10
Q_3	1/0.97	2.3	1
Q_4	-	-	1
Q_5	0.98/1	11	8
Q_6	1/1	73	8
Q_7	0.64/1	70	28
Q_8	0.28/1	170	32

Table 3: Comparison between DP and Greedy

Cardinality value	Q_1'	Q_1''
4	0.8	0.56
5	1	0.65
6	1	0.71
7	1	0.75

Table 4: The relative aggregated value of Greedy over the optimal solution

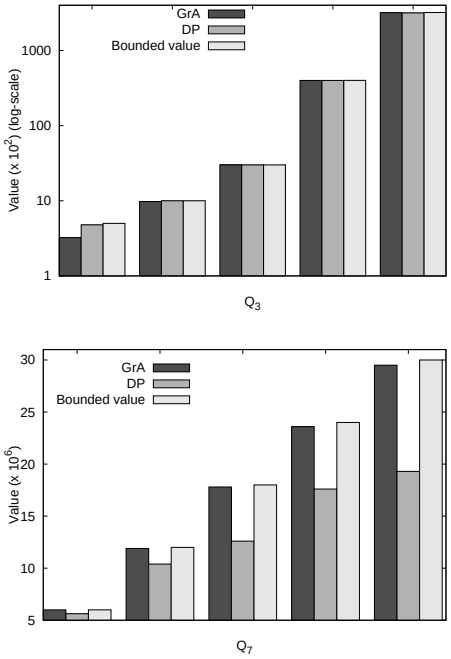


Figure 3: Optimal values obtained by DP and Greedy

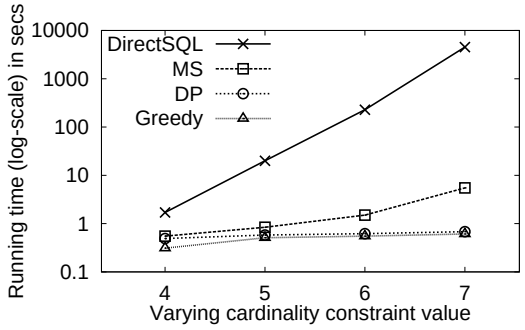
Running Time. The third and fourth columns in Table 3 show the running time comparison between Greedy and DP, where Greedy runs 1.5 - 9 times faster than DP. The result is expected since Greedy is a heuristic solution.

6.2 Comparing DP, Greedy, MS and DirectSQL

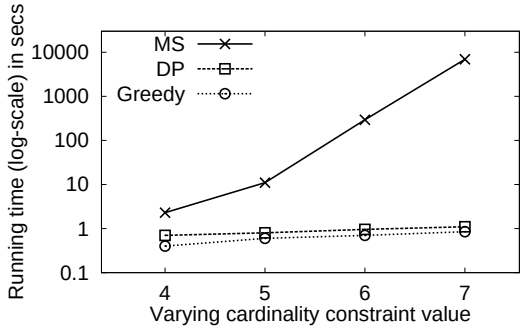
In this section, we compare our proposed methods (DP and Greedy) against the direct approach that uses SQL queries, denoted by DirectSQL, and the method MS proposed in [8] for bounded SAC queries, which have a bounded cardinality constraint to bound the cardinality of the qualified sets. For this comparison, we used two queries Q_1' and Q_1'', which are variants of query Q_1. The three methods (DP, MS, and DirectSQL) are able to find the optimal results for both queries Q_1' and Q_1''.

First, we run these approaches for query Q_1' by varying the set cardinality constraint value from 4 to 7; i.e., $x \in [4, 7]$. The Greedy approach returns rather good results; i.e., only for the cardinality 4, the relative aggregated value of Greedy over the optimal solution is 0.8; for the other cases, Greedy achieves the optimal aggregated values (the second column in Table 4). In terms of the running time, as shown in Figure 4(a), Greedy runs the most efficiently. For the other three methods, DP runs faster than MS and much faster than the direct approach (DirectSQL). Note that the number of tuples selected by the base query in Q_1' is controlled to be small (i.e., 60) so that the DirectSQL approach can complete its evaluation within reasonable time. Observe that when the set cardinality value increases, the running times of DP and Greedy increase linearly; whereas the running times of MS and DirectSQL increase exponentially.

Second, we increase the maximum sum value in Q_1 to be 1000 and the number of selected tuples by the base query to be 500 to scale up the size of the matrices built by DP and the



(a) Q_1'



(b) Q_1''

Figure 4: The running time comparison between DP, Greedy, MS and DirectSQL

number of subsets to be considered by MS and DirectSQL. We denote this new query by Q_1''. For Q_1'', DirectSQL cannot finish in five hours, thus we do not record the results of DirectSQL in Figure 4(b). The relative aggregated value of Greedy over the optimal solution is in the range of 0.56 to 0.75 (the third column in Table 4). In terms of the running time as shown in Figure 4(b), Greedy still runs the most efficiently. DP runs much more efficiently than MS for this query, since the number of subsets considered by MS is large. For instance, for the case with *cardinality* = 6, the number of subsets of size six considered by MS is 3800000 and the running time is 5 minutes.

6.3 Summary

We have the following conclusions from our experimental study:

- DP can find optimal solutions for multi-sum SAC queries when its memory requirement is within the available main memory with a trade-off of slower running time compared to Greedy in the order of 1.5 - 9 times.
- The solutions found by Greedy have high quality and are better than these by DP when the memory requirement of DP exceeds the available main memory. Furthermore, Greedy runs very efficiently.
- For bounded SAC queries, both DP and Greedy run much more efficiently than the direct method (DirectSQL) of using SQL queries and MS method proposed for preferences involving fixed-cardinality sets [8].

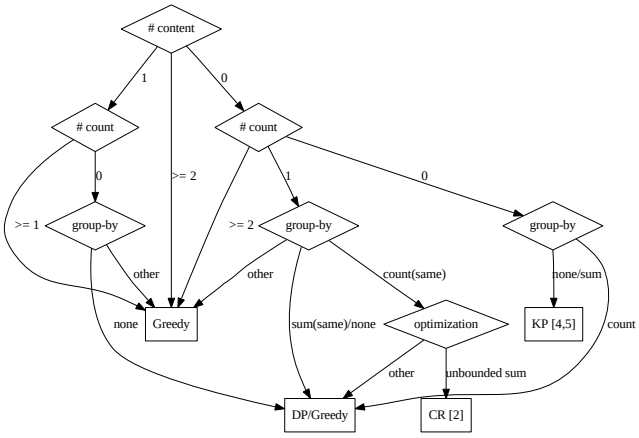


Figure 5: Evaluation Strategies for SAC Queries

7. RELATED WORK

The related work can be classified into two categories: work that are directly related to SAC queries [2, 4, 5] whose relationship to our work (DP and Greedy) is captured by the decision tree in Figure 5, and work that are related more broadly to set-based queries [1, 3, 7, 8] which involved constraints different from SAC queries.

SAC Queries. Several variants of SAC queries that have been studied differ in the number and types of constraints permitted. These work together with our techniques can be classified by the decision tree in Figure 5, where each leaf node represents the best evaluation technique for the fragment of SAC queries represented by the path of internal nodes which specify the permitted constraints. Specifically, "# contents" specifies the number of content constraints supported (0, 1, or ≥ 2), "# count" specifies the number of count constraints supported (0, 1, or ≥ 2), "group-by" specifies the type of group-by constraints supported (count, sum, none) and whether the attribute(s) used with group-by operators are the same (indicated by "same") or not (no explicit label) with the attribute(s) used with the count constraints, and "optimization" specifies the type of optimization constraints supported (unbounded sum, other).

KP refers to the classic knapsack problem techniques [5]: Given a set of items where each item j has a profit p_j and a weight w_j, the goal is to select a subset of items such that its total profit is maximized and its total weight does not exceed an input capacity value. A variant of KP was studied in [4] for solving optimization under parametric aggregation constraints (OPAC) query, which takes the following as inputs: (1) a relation $R(A_1, \cdots, A_n, P)$, (2) a set of parametric sum-aggregation constraints of the form $sum(A_i) \leq c_i$ with c_i as a parameter, and (3) a sum optimization constraint $sum(P)$ to be maximized. Given a parameterized OPAC query, [4] proposed an algorithm to construct indices to efficiently provide approximate answers with guarantee bound on its accuracy for any instantiated OPAC query with specific values for the parameters in the sum constraints.

A second variant of KP, which corresponds to the fragment of SAC queries with multiple sum and a single group-by-sum constraints, is the *multiple-choice* Knapsack problem [5]. This is useful in budgeting applications to select a

set of projects to be funded such that the total cost for all projects is bounded by some limit, the total cost for projects belonging to the same department is bounded by another limit, and the total project profit is maximized. This problem can be solved in pseudo-polynomial time using a two-step dynamic programming approach [5] which is similar to our proposed DP. However, the formulation of the second dynamic programming stage in our DP is more complicated, since DP needs to take into account the count/content constraints, which [5] does not consider.

Another related work is the CourseRank (CR) project [2] which is motivated by course planning applications. CR considers constraints of the form "take at least a and at most b courses from a set S_i", where a and b are non-negative integers and S_i is a set of related courses (e.g. CS courses), and each course is associated with a use-preference score. For example, a student might be required to complete 2 or 3 courses from a given set of six math courses. Given such constraints, CR finds a set of courses that satisfies all requirements such that the number of selected courses is equal to some given value and the total score of the selected courses is maximized. A polynomial-time algorithm based on maximal flow was proposed for the CR problem [2].

Set-based Queries. There are also several work on set-based query evaluation [1, 3, 7, 8] but they differ from our work due to the types of constraints supported in the queries and/or the focus of the evaluation problem.

A related work, which is motivated by online shopping applications [1], examines the problem of recommending a set of "satellite items" (e.g., case, speaker) related to a given "center item" (e.g., iPhone). Given a budget B and a central item, [1] finds (approximately) all *maximal* sets of satellite items associated with the central item such that the cost of each maximal set does not exceed the given budget B. Different from [1], we do not consider "maximal set" constraints, which will make the problem of evaluating SAC queries even harder. However, SAC queries support other constraints (e.g., count, group-by, content) which [1] does not handle.

The work on making composite recommendations of a set of items is studied in [7], where each item is associated with both a rating value and a cost. The goal is to find the top-k sets of items such that the total cost of items in each set is no greater than a given budget, and a set with a higher total rating value is more preferable. Note that when $k = 1$, the problem setting is exactly the Knapsack problem. The problem addressed there, however, is based on an evaluation framework where the ratings of items are accessed via sorted access from some external recommending parties, while the costs of items are accessed via random access. [7] introduced a 2-approximation solution and a greedy algorithm to find top-k composite recommendations with the optimization goal to minimize the total access cost.

The work in [3] finds an optimal subset of a set of tuples according to a set preference, which is specified as either a TCP-net or a scoring function on a collection of set properties. A set property is based on the number of tuples satisfying a certain predicate (e.g., the number of tuples satisfying a predicate is greater than a given value). Two heuristic search algorithms (based on branch-and-bound and Constraint Satisfaction Problem) were proposed in [3]. There are two main differences between [3] and our work. First, [3] focuses on cardinality constraint and does not support other types of constraints considered in our work. Second, the algorithms developed for each work are different from the other.

Finally, there is also a recent related work on finding all subsets of a given fixed-cardinality from a relation that satisfy some user-specified preferences [8]. There are two main differences between the work in [8] and ours. First, [8] deals with sets of *fixed-cardinality*, whereas the sets retrieved by SAC queries can have varying cardinality. Second, [8] allows users to specify preferences over sets and the focus is on computing the skyline query result. However, as explained in Section 5, the technique in [8] can be applied to evaluate SAC queries with a fixed set cardinality.

In summary, although several related work have examined set-based queries or special fragments of SAC queries, our work on DP is the first pseudo-polynomial time algorithm for evaluating the non-trivial SAC query fragment with multiple sum and at most one of either count, group-by, or content constraint. Furthermore, our Greedy approach is the first heuristic to evaluate general SAC queries with any combination of count, sum, group-by, and content constraints. The Greedy heuristic tries to find a solution that satisfies all the specified constraints but may return an approximate solution that meets only some of the constraints.

8. CONCLUSION

In this paper, we have examined the evaluation of set-based queries with aggregation constraints (SAC queries), which are very useful for many data retrieval applications. We have presented two novel algorithms: DP, a pseudo-polynomial time algorithm for evaluating a non-trivial fragment of SAC queries involving multiple sum constraints and at most one of count, group-by, or content constraint; and Greedy, the first heuristic approach for evaluating general SAC queries. The effectiveness of our proposed solutions was demonstrated by an experimental performance study over real data sets. As part of our future work, we plan to integrate ranking/top-k pruning into SAC query evaluation.

Acknowledgements This research is supported in part by NUS Grant R-252-000-453-112.

9. REFERENCES

[1] S. Basu Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu. Constructing and exploring composite items. In *SIGMOD*, pages 843–854, 2010.

[2] B. Bercovitz, F. Kaliszan, G. Koutrika, H. Liou, A. Parameswaran, P. Venetis, Z. M. Zadeh, and H. Garcia-Molina. Social sites research through courserank. *SIGMOD Rec.*, 38(4), 2009.

[3] M. Binshtok, R. I. Brafman, S. E. Shimony, A. Martin, and C. Boutilier. Computing optimal subsets. In *AAAI*, pages 1231–1236, 2007.

[4] S. Guha, D. Gunopoulos, N. Koudas, D. Srivastava, and M. Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB*, pages 778–789, 2003.

[5] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[6] Q. T. Tran, C.-Y. Chan, and G. Wang. Evaluation of set-based queries with aggregation constraints. Technical Report http://www.comp.nus.edu.sg/~tqtrung/sac-tech.pdf, National University of Singapore, August 2011.

[7] M. Xie, L. V. Lakshmanan, and P. T. Wood. Breaking out of the box of recommendations: from items to packages. In *RecSys*, pages 151–158, 2010.

[8] X. Zhang and J. Chomicki. Preference queries over sets. In *ICDE*, pages 1019–1030, 2011.