Efficient Processing of Enumerative Set-based Queries

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

ABSTRACT

Many applications often require finding sets of entities of interest that meet certain constraints. Such set-based queries (SQs) can be broadly classified into two types: *optimization SQs* that involve some optimization constraint and *enumerative SQs* that do not have any optimization constraint. While there has been much research on the evaluation of optimization SQs, there is very little work on the evaluation of enumerative SQs, which represent the most fundamental fragment of set-based queries. In this paper, we address the problem of evaluating enumerative SQs using RDBMS. While enumerative SQs can be expressed using SQL, existing relational engines, unfortunately, are not able to efficiently evaluate such queries due to their complexity. In this paper, we propose a novel evaluation approach for enumerative SQs. Our experimental results on PostgreSQL demonstrate that our proposed approach outperforms the conventional approach by up to three orders of magnitude.

1. INTRODUCTION

Many applications often require finding sets of entities of interest that meet certain constraints. Such set-based queries (SQs) can be broadly classified into two types: *optimization SQs* that involve some optimization constraint and *enumerative SQs* that do not have any optimization constraint. For example, consider a relation *R(id,type,city,price,duration,rating)* shown in Table 1 that stores information about various places of interest (POI), where *type* refers to the category of the POI (e.g., museum, park), *duration* refers to the recommended duration to spend at the POI and *rating* refers to the average visitors' rating of the POI. Suppose that a tourist is interested to find all tour trips near Shanghai consisting of POIs that meet the following constraints: the trip must include both Shanghai (S.H.) and Suzhou (S.Z.) cities, the trip must include POIs of type museum and park, and the total duration of the trip should be between 6 and 10 hours. There are three packages that satisfy the above query: $\{t_1, t_2\}$, $\{t_1, t_2, t_3\}$ and $\{t_1, t_2, t_5\}$. The above is an example of an enumerative SQ to find all sets of POIs that satisfy the given constraints. If the query had an additional constraint to minimize the total cost of the tour package, it would become an optimization SQ.

As another example, suppose that an employer is looking to hire a team of language translators for a project that meet the following constraints: each team member must know English; the team collectively must be knowledgeable in French, Russian, and Spanish; the team consists of at least two translators; and the total monthly salary of the team is no more than $\$50K$. Consider a relation *Translator (id,location,salary,english,french,russian,spanish)* that stores information about language translators available for hire, where the four binary valued attributes *english*, *french*, *russian*, and *spanish* indicate whether a translator is knowledgeable in the

Table 1: An example relation R

id	type	city	price	duration	rating
t_1	museum	S.H.	50	4	7
t_2	park	S.Z.	70	3	5
t_3	museum	S.Z.	60	3	8
t_4	shopping	S.H.	80	5	7
t_5	shopping	H.Z.	90	2	9

specific languages, *location* represents the translator's living place, and *salary* represents the translator's expected monthly salary. To browse through all the possible teams for hiring, the employer executes an enumerative SQ on the *Translator* relation.

Another application of enumerative SQs is in the area of set preference queries [1, 2, 3], which computes all sets of entities of interest that satisfy some preference function. Consider again our example on hiring translators. In addition to the previously discussed constraints, the employer could prefer to hire a team where (a) the team members are located close to one another and (b) their total salary is low. Thus, this set preference query is essentially a skyline [4] set-query to retrieve non-dominated teams where the members have close proximity and low total salary. The most general approach to evaluate skyline set-queries is to first enumerate all the candidate sets followed by pruning away the dominated sets. Although there has been recent work to integrate these two steps [3], such optimization is applicable only for restricted cases (e.g., when the sets are of fixed cardinality and the preference function satisfies certain properties); and is not applicable for queries such as our example query. Therefore, efficient algorithms to evaluate enumerative SQs are essential for the efficient processing of set preference queries.

There has been much research on evaluating optimization SQs where the focus is on heuristic techniques to compute approximately optimal or incomplete query results (e.g., [5, 6, 7, 8, 9, 3, 10]). However, to the best of our knowledge, there has not been any prior work on the evaluation of enumerative SQs. Enumerative SQs are essentially a generalization of conventional selection queries to retrieve a collection of sets of tuples (instead of a collection of tuples), and they represent the most fundamental fragment of set-based queries.

In this paper, we address the problem of evaluating enumerative SQs using RDBMS. For convenience, we refer to enumerative SQs as simply SQs in the rest of this paper.

While SQs can be expressed using SQL, existing relational engines, unfortunately, are not able to efficiently optimize and evaluate such queries due to their complexity involving multiple self-joins and/or view expressions. In this paper, we propose a novel evaluation approach for SQs. The key idea is to first partition the in-

put relation into disjoint blocks based on the different combinations of constraints satisfied by the tuples and then compute the answer sets by appropriate combinations of the blocks. In this way, a SQ is evaluated as a collection of cross-product queries (CPQs). However, applying existing multiple query optimization (MQO) techniques for this evaluation problem is not effective for two reasons. First, the scale of the problem could be very large involving hundreds of CPQ evaluations. Existing MQO heuristics, which are mainly designed for optimizing a handful of queries, are not scalable for our problem. Second, as the queries here are CPQs (and not join queries), existing MQO techniques, which are based on materializing and reusing common subexpressions, is not effective as the cost of materialization exceeds the cost of recomputation.

In this paper, we make three key contributions to the study of SQs. First, we experimentally show that conventional RDBMS are unable to efficiently evaluate SQs. Second, we propose a novel approach to evaluate SQs in terms of a collection of CPQs. Our approach includes both effective MQO heuristics designed to optimize a large collection of CPQs and efficient evaluation techniques that exploit the properties of set predicates in the SQs. Third, we demonstrate the effectiveness of our approach with a comprehensive experimental evaluation on PostgreSQL which shows that our approach outperforms the conventional SQL-based solution by up to three orders of magnitude.

The rest of this paper is organized as follows. In Section 2, we formally introduce set-based queries (SQs) and a fragment of SQs referred to as basic SQs (BSQs). Section 3 presents some preliminaries. Section 4 presents a baseline SQL-based solution to evaluate SQs. Section 5 presents our main-memory based approach to evaluate BSQs, and Section 6 extends the approach to evaluate BSQs on disk-based data. In Section 7, we extend our approach to evaluate general SQs beyond BSQs. Section 8 presents an experimental performance evaluation of the proposed techniques. Section 9 presents related work, and we conclude our paper in Section 10.

2. SET-BASED QUERIES

In the simplest form, a *set-based query (SQ)* Q is defined by an input relation R, which represents a collection of entities of interest, and an input set of predicates P on R. The query's result is a collection of all the subsets of R such that each subset satisfies the predicates in P.

For convenience, we introduce an extended SQL syntax to express SQs more explicitly. The example SQ in Section 1 can be expressed by the following extended SQL query.

Q_{poi}: SELECT *
FROM SET(R) S
WHERE v_1 in S AND v_2 in S
AND v_3 in S AND v_4 in S
AND v_1.city = S.H. AND v_2.city = S.Z.
AND v_3.type = museum AND v_4.type = park
AND $6 \leq$ SUM(S.duration) ≤ 10

The "SET(R) S" in the from-clause specifies S as a *set variable* whose value is a subset of tuples in relation R. Each of the predicates of the form "v_i in S" specifies v_i as a *member variable* representing a member of the set variable S. Note that the values of the member variables are not necessarily distinct. Each of the next four predicates specifies a constraint on an individual member; and the last predicate specifies an aggregation constraint on the set. The output schema of this query consists of all the attributes in relation R and an additional, implicit integer attribute named sid

Table 2: Output of the example SQ

sid	id	type	city	price	duration	rating
1	t_1	museum	S.H.	50	4	7
1	t_2	park	S.Z.	70	3	5
2	t_1	museum	S.H.	50	4	7
2	t_2	park	S.Z.	70	3	5
2	t_3	museum	S.Z.	60	3	8
3	t_1	museum	S.H.	50	4	7
3	t_2	park	S.Z.	70	3	5
3	t_5	shopping	H.Z.	90	2	9

Table 3: Classification of commonly used set predicates

Set Predicates	Anti-monotone	Monotone
$MIN(S.A) \geq c$	yes	no
$MIN(S.A) \leq c$	no	yes
$MAX(S.A) \geq c$	no	yes
$MAX(S.A) \leq c$	yes	no
$SUM(S.A) \geq c, \forall t \in S, t.A \geq 0$	no	yes
$SUM(S.A) \leq c, \forall t \in S, t.A \geq 0$	yes	no
$COUNT(S) \geq c$	no	yes
$COUNT(S) \leq c$	yes	no
$AVG(S.A)\ \theta\ c, \theta \in \{\geq, \leq\}$	no	no

that represents the identifier for an answer set. The values of sid are generated automatically by the database system. The attributes (sid, id) form the key of the output schema where id is the key of input relation R. Thus, each answer set to the query is represented by a collection of output tuples having the same sid value. Table 2 shows the output of the example SQ Q_{poi}.

There are two types of selection predicates in a SQ. A *member predicate* specifies a constraint on exactly one member variable (e.g., "v_1.city = S.H."). A *set predicate* specifies a constraint on a set variable or more than one member variable; examples include "SUM(S.duration) $\leq$ 10" and "v_1.price + v_3.price $\leq$ 100".

Given a set predicate p, it is classified as *anti-monotone* if whenever a set S does not satisfy p, then any superset of S also does not satisfy p; it is classified as *monotone* if whenever a set S satisfies p, then any superset of S also satisfies p. In our example SQ Q_{poi}, the predicate "SUM(S.duration) $\leq$ 10" is an anti-monotone set predicate, while the predicate "SUM(S.duration) $\geq$ 6" is a monotone set predicate. An example of a set predicate that is neither monotone nor anti-monotone is "AVG(S.price) $\leq$ 20". Note that set predicates can also involve other SQL constructs such as *groupby-clause* and *having-clause* which we omit in this paper. Table 3 shows a classification of commonly used set predicates in SQL. Anti-monotone set predicates have the property that a conjunction/disjunction of anti-monotone set predicates is anti-monotone. Similarly, monotone set predicates have the property that a conjunction/disjunction of monotone set predicates is monotone. If the monotone/anti-monotone property of a set predicate in a query is unknown, our query processing approach would conservatively assume that the predicate in question is neither anti-monotone nor monotone and hence does not apply any monotone/anti-monotone specific optimizations.

The minimum/maximum cardinality of an answer set is determined as follows. If the query explicitly specifies an upper bound on the the set's cardinality (e.g., "COUNT(S) $\leq$ 3"), then the maximum cardinality is given by specified constraint; otherwise, the maximum cardinality is the number of member variables in the query. If the query explicitly specifies a lower bound on the the set's cardinality (e.g., "COUNT(S) $\geq$ 2"), then the minimum cardinality is given by specified constraint; otherwise, the minimum

cardinality is equal to one.

Since the number of qualifying answer sets could be very large for some SQs, there are two natural ways to limit the size of the query result. The first approach is to retrieve only some fixed number of say k result sets either using a limit clause to retrieve any k sets or via a ranking function to retrieve the top-k sets. The second approach is to retrieve only *minimal sets* that satisfy the query's predicates. A set S is defined to be minimal if no proper non-empty subset of S also satisfies the predicates in P. For example, the answer sets $\{t_1, t_2, t_3\}$ and $\{t_1, t_2, t_5\}$ for the example SQ Q_{poi} are not minimal since their subset $\{t_1, t_2\}$ also satisfies the query's predicates. Minimal sets are also of interest on their own as they serve as a concise representation of all the answer sets (i.e., any superset of a minimal answer set is also an answer set) if all the set predicates in the query are monotone. Furthermore, there are use cases where minimal sets represent the desired answer sets. As an example, consider a graduating student who is deciding on the set of courses to enroll for her final semester of study. To meet her school's graduation requirement, she still need to complete at least 16 modular credits of courses from the Computer Science department of which at least two courses must be at the 4000 level. For this scenario, minimal answer sets would be an appropriate fit for the user's intention; moreover, the number of non-minimal answer sets is probably too many to browse. The minimal set constraint can be expressed in our extended SQL syntax by replacing "SET(R) S" by "MINSET(R) S" to indicate that S is a *minimal set variable*.

To simplify the presentation of evaluation algorithms for SQs, we introduce a special fragment of SQs called *basic SQs*. A SQ Q is defined to be a basic SQ (BSQ) if Q retrieves only minimal sets and all the set predicates in Q are anti-monotone[1]. BSQs have the following property: if a tuple t in R does not satisfy any member predicate in a BSQ Q, then t will not contribute to any answer set of Q. The reason for this is as follows. If there exists an answer set S containing t, then since all the set predicates in a BSQ are anti-monotone, $S \setminus \{t\}$ would also satisfy all the set predicates in Q. This implies that $S \setminus \{t\}$ is an answer set which contradicts the fact that S is a minimal set. Note that this property of BSQs does not hold for general SQs. For example, $\{t_1, t_2, t_5\}$ is an answer set for our example SQ Q_{poi}, which contains the tuple t_5 that does not satisfy any member predicate. However, if we restrict Q_{poi} to retrieve only minimal answer sets, then t_5 would not contribute to any answer set of Q_{poi}.

We should emphasize that the focus of this paper is not on the design of SQL extensions but on efficient query evaluation. The above example is meant to illustrate how the semantics of SQs can be expressed more explicitly and easily using some SQL extensions instead of using conventional SQL, which we will discuss in Section 4.

3. PRELIMINARIES

In this paper, we consider a SQ Q defined over a relation R, where there are n member variables in Q. Without loss of generality, we assume that the maximum cardinality of the answer sets for Q is n.

Let $V = \{v_1, \cdots, v_n\}$ denote the set of member variables in Q. The predicates P in Q can be partitioned into $n+1$ subsets, P_0, $P_1, \cdots, P_n$, where each P_i, $i \in [1, n]$, denote the set of member predicates in Q that involves the member variable v_i; and P_0 denote the set of set predicates in Q.

In this paper, we refer to a set S as a *k-set* to mean that the cardinality of S is k. Thus, each answer set for Q is an i-set, where $i \in [1, n]$.

EXAMPLE 1. *In our example SQ Q_{poi}, there are four member variables (i.e., v_1, v_2, v_3 and v_4). Therefore, the predicates can be partitioned into five subsets: $P_0 = \{6 \leq SUM(S.duration) \leq 10\}$, $P_1 = \{v_1.city = S.H.\}$, $P_2 = \{v_2.city = S.Z.\}$, $P_3 = \{v_3.type = museum\}$ and $P_4 = \{v_4.type = park\}$.* □

4. BASELINE SOLUTION USING SQL

In this section, we outline a baseline approach to evaluate SQs using conventional SQL. The detail illustration of the baseline approach is given in A.

In this approach, answer sets are generated iteratively, i.e., answer i-sets are computed before answer $(i+1)$-sets, which is similar to the Apriori-style of using SQL to compute frequent itemsets [11]. Let C_i $(1 \leq i \leq n)$ denote the collection of candidate answer i-sets that satisfy all the anti-monotone set predicates in P_0, and $A_i \subseteq C_i$ denote the collection of answer i-sets. Each C_i/A_i is represented by a relation/view where each tuple in C_i/A_i represents a subset of i tuples from R. Each C_i, $i \geq 2$, is computed using a self-join of C_{i-1} and each A_i is derived from C_i. In this approach, the answer sets for a SQ are given by multiple output tables $A_1, \cdots, A_n$, where each tuple in each A_i presents an answer i-set for Q.

In the first iteration, C_1 is the subset of tuples in R that satisfy all the anti-monotone set predicates in P_0. A_1 is the subset of tuples in C_1 that satisfy all the predicates in Q. In the ith iteration, $i > 1$, C_i is computed by a self join of C_{i-1} to ensure two requirements. First, C_i does not contain duplicate candidate answer i-sets. Following the same principle to avoid duplicates in [11], the self-join of C_{i-1} to compute C_i has $(i-2)$ equi-join predicates requiring that two matching tuples in C_{i-1} (representing two $(i-1)$-sets) have $(i-2)$ identical tuples. Second, each tuple in C_i satisfies all the anti-monotone set predicates in P_0. A_i is derived from C_i by appropriate selection predicates to ensure that each tuple in A_i satisfies all the predicates in Q. Thus, this approach is implemented as a sequence of SQL queries where the number of queries is a linear function of n (details are given in A).

EXAMPLE 2. *Figure 1 illustrates the first two iterations of the baseline approach for evaluating our example SQ Q_{poi} on the input relation R in Table 1. To avoid clutter, the non-relevant attributes (i.e., $price$ and $rating$) are omitted from the figure. In the first iteration, C_1 is computed by Q_1 on R to ensure that each tuple in C_1 (representing a candidate answer 1-set) satisfies all the anti-monotone set predicates. The answer 1-sets are given by A_1 which is computed by Q_2 on C_1; A_1 is empty since there is no answer 1-set for this SQ. In the second iteration, C_2 is computed by Q_3 with a self-join on C_1 and A_2 is computed from C_2 using Q_4. Observe that A_2 contains one answer 2-set $\{t_1, t_2\}$. Since the answer sets for this query has a maximum cardinality of four, this process continues for two additional iterations to find answer 3- and 4-sets (details not shown).*

Minimal set constraint. If the query requires only minimal answer sets, then the above approach still works with the following two extensions. First, to generate C_i (representing candidate answer i-sets), the self join is performed on $C_{i-1} \setminus A_{i-1}$ instead of C_{i-1} as all the supersets of answer $(i-1)$-sets in A_{i-1} are not minimal. Second, for each tuple in A_i, in addition to satisfying all

[1] By definition of BSQs, the maximum cardinality of a BSQ's answer sets is bounded by the minimum of the number of member variables in the query and any explicitly specified cardinality constraint.

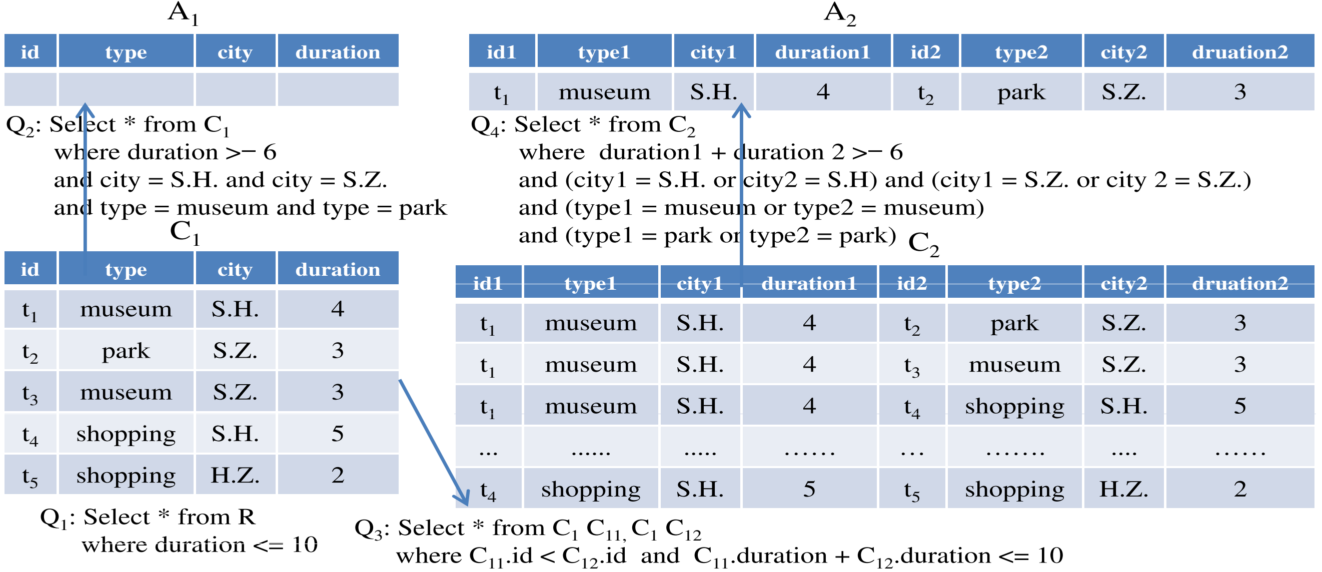


Figure 1: Illustration of the first two iterations of the baseline SQL-based solution

the predicates in Q, it must also represent a minimal set. To verify the minimality of a candidate answer i-set $S \in C_i$, all the subsets of S have to be examined to ensure that they do not satisfy all the predicates in Q. However, if P_0 contains only anti-monotone and monotone set predicates, then only subsets with a cardinality of $(i-1)$ need to be examined.

Alternative SQL-based approach for BSQs. For BSQs, there is an alternative SQL-based approach that generates all the answer sets in a single output table with arity equal to the maximum cardinality of the answer sets given by n. This approach consists of two main steps. The first step generates all the candidate answer sets in a relation/view M by computing the cartesian product of n views $M_1, \cdots, M_n$, where each M_i is the set of tuples in R that satisfies P_i. Note that M may contain multiple tuples that represent the same candidate answer set since each tuple in R may appear in multiple M_i's. Therefore, we need to remove the duplicate candidate answer sets from M. The second step computes the answer sets by eliminating those candidate answer sets in M that are duplicates, do not satisfy P_0, or are not minimal. The details of this approach are given in A.

It is important to note that this alternative approach is not applicable for evaluating SQs since a tuple from R can contribute to an answer set even if it does not appear in any M_i $(1 \le i \le n)$ as discussed in Section 2. For evaluating BSQs, our experimental results show that the alternative approach is significantly outperformed by the first discussed approach. The main reason is due to the complex SQL queries used to remove duplicate and non-minimal candidate answer sets in the second step. Given its limited applicability and poor performance, we will not consider the alternative approach any further in this paper.

5. BASIC APPROACH

To simplify the presentation of evaluation algorithms for SQs, we first present the evaluation of BSQs in this section assuming that all the data and structures can be stored in main memory, and then describe the extensions to handle large, external data in Section 6. We extend our techniques for (general) SQs beyond BSQs in Section 7.

Recall that a BSQ Q retrieves only minimal sets and all the set predicates in Q are anti-monotone. Our proposed approach evaluates a BSQ Q in two phases. In the first phase, a sequential scan of R is performed to partition R into s disjoint, non-empty blocks, $R_{V_1}, \cdots, R_{V_s}$, $s \in [1, 2^n]$, where each $V_i \subseteq V$ is a subset of member variables in Q, and $R_{V_i} \subseteq R$ represents the tuples that satisfy exactly all the member predicates (i.e., $\bigcup_{v_j \in V_i} P_j$) associated with the member variables in V_i. Specifically, R is partitioned as follows: for each tuple t in R, t belongs to block R_{V_i} where $V_i \subseteq V$ is the subset of all member variables such that for each member variable $v \in V_i$, t satisfies all the member predicates involving v. Thus, for each member variable $v \in V \setminus V_i$, t does not satisfy some member predicate involving v.

There are two blocks of R, namely, $R_\emptyset$ and R_V, that are not materialized during the partitioning phase [2]. The block $R_\emptyset$ contains tuples in R that do not satisfy any P_i $(1 \le i \le n)$ in Q. For a BSQ Q, none of the tuples in $R_\emptyset$ will contribute to an answer set. Therefore, the block $R_\emptyset$ is not materialized during the partitioning. At the other extreme, each tuple in R_V satisfies all P_i $(1 \le i \le n)$ in Q; therefore, each tuple in R_V forms an answer 1-set if it also satisfies P_0. If a tuple in R_V does not satisfy P_0, it will not contribute to any answer set for a BSQ and can be ignored. Since each tuple in R_V can be either directly output as an answer set or ignored, these tuples will not contribute to additional answer sets; thus, this block is also not materialized during partitioning. The blocks materialized in the first phase will be used in the second phase to generate further answer sets.

In the second phase, the remaining answer sets are generated by combining tuples from appropriate blocks such that the combined set of tuples qualifies as an answer set; i.e., the set of tuples is a minimal set of tuples that satisfies all the query's predicates. Each such combination of blocks is then evaluated as a cross-product query (CPQ); thus, the remaining answer sets are computed as a union of CPQs. To enumerate these answer sets, we first need to characterize the appropriate combinations of block sets.

Consider a set of blocks $U = \{R_{V_1}, \cdots, R_{V_k}\}$ where each

[2]For SQs, both $R_\emptyset$ and R_V have to be materialized as discussed in Section 7.1.

$R_{V_i} \neq \emptyset$. Note that U is not necessarily a partition of R; i.e., $R_{V_1} \cup \cdots \cup R_{V_k} = R$. We define U to be a *valid block set (or vbset)* if U satisfies the following two properties: (P1) $\bigcup_{R_{V_i} \in U} V_i = V$; and (P2) no proper subset of U satisfies P1. Property 1 ensures that a candidate answer set S formed by selecting one member from each block in U will satisfy all the member predicates in Q, while property 2 ensures that S is minimal.

For convenience, we refer to a vbset that is a k-set as a k-vbset. We use *VBSet* to denote the collection of all vbsets.

Thus, if $U = \{R_{V_1}, \cdots, R_{V_k}\}$ is a k-vbset, then a k-set $S = \{t_1, \cdots, t_k\}$, where $t_i \in R_{V_i}$, $i \in [1, k]$, is an answer set for Q if S satisfies P_0. Therefore, the remaining answer sets for Q is computed by evaluating a collection of CPQs, where each CPQ is associated with a vbset.

Our overall approach evaluates Q based on the following expression:

$$\sigma_{P_0}(R_V \cup \bigcup_{U_i \in VBSet} (\bigtimes_{R_{V_j} \in U_i} R_{V_j}))$$

$\sigma_{P_0}(R_V)$ is evaluated in the first phase while $\sigma_{P_0}(\bigcup_{U_i \in VBSet} (\bigtimes_{R_{V_j} \in U_i} R_{V_j}))$ is evaluated in the second phase. The cross-product expression represents a CPQ corresponding to the vbset U_i, the union expression enumerates all the vbsets[3], and the final selection operator selects the minimal sets that satisfy all the set predicates in P_0.

EXAMPLE 3. *Consider the evaluation of the BSQ Q'_{poi} that is derived from our example SQ Q_{poi} by removing its non-anti-monotone set predicate (i.e., $SUM(S.duration) \geq 6$). In the first phase, for the tuple t_1, since it exactly satisfies the member predicates (i.e.,$P_1 = \{v_1.city = S.H.\}$ and $P_3 = \{v_3.type = museum\}$) associated with the member variables v_1 and v_3, it is put into the block $R_{\{v_1,v_3\}}$. Following the same approach for the remaining tuples, R is partitioned into five blocks: $R_{\{v_1,v_3\}} = \{t_1\}$, $R_{\{v_2,v_4\}} = \{t_2\}$, $R_{\{v_2,v_3\}} = \{t_3\}$, $R_{\{v_1\}} = \{t_4\}$ and $R_\emptyset = \{t_5\}$. Note that the block $R_\emptyset$ is not materialized. In the second phase, two vbsets, $\{R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}\}$ and $\{R_{\{v_1\}}, R_{\{v_2,v_3\}}, R_{\{v_2,v_4\}}\}$, are enumerated which generate two candidate answer sets $\{t_1, t_2\}$ and $\{t_2, t_3, t_4\}$. Among them, only $\{t_1, t_2\}$ satisfies the anti-monotone set predicate (i.e., $SUM(S.duration) \leq 10$) and forms an answer set.*

In the following, we elaborate on the details of the second phase, namely, how to efficiently enumerate vbsets and evaluate the corresponding CPQs.

Enumeration of vbsets. Given the blocks of R created in the first phase, the collection of all vbsets $VBSet$ is efficiently enumerated based on the following proposition.

PROPOSITION 1. *If U is a k-vbset, then it satisfies the following three properties: (1) for each $R_{V_i} \in U$, the cardinality of V_i is at most $n - k + 1$; (2) there must exist a block $R_{V_i} \in U$ such that the cardinality of V_i is at least $\lceil \frac{n}{k} \rceil$; (3) for any pair of distinct blocks R_{V_i} and R_{V_j} in U, $V_i \not\subseteq V_j$ and $V_j \not\subseteq V_i$.*

The proof of the proposition is given in Appendix B. Based on the proposition, we enumerate all the vbsets by computing the cartesian product of n sets (with the above three properties enabled to prune the cartesian product space) where each set is $\{R_{V_1}, \cdots, R_{V_s}\}$ representing the set of all generated blocks in the partitioning phase.

[3]The union operator is used only to combine the results and not to eliminate duplicates as the generated results are all unique.

Thus, the time complexity to enumerate all the vbsets is $O(2^{n^2} n^2)$ where $O(2^{n^2})$ is the time complexity to compute the cartesian product to generate all the candidate vbsets and $O(n^2)$ is the time complexity to determine a candidate vbset is indeed a vbset. As the value n is not expected to be large for BSQs, it is very fast to enumerate all the vbsets by exploiting the above three properties.

EXAMPLE 4. *Continuing with Example 3. Here we have $n = 4$. From the first property, block $R_{\{v_1,v_3\}}$ will not form a 4-vbset since the cardinality of the set $\{v_1, v_3\}$ is 2. From the second property, for a 2-vbset, at least one block should satisfy two P_i $(1 \leq i \leq 4)$, otherwise the 2-vbset can not satisfy all P_i $(1 \leq i \leq 4)$. From the third property, blocks $R_{\{v_1,v_3\}}$ and $R_{\{v_1\}}$ will not appear in the same vbset since one is a subset of the other.*

Evaluation of CPQs. Each CPQ is evaluated using a *multi-way nested-loop cross-product* (MNLCP) approach, which is a generalization of the well-known binary nested-loop join algorithm. For convenience, we use the notation $(R_{V_1}, \cdots, R_{V_k})$ to refer to a CPQ Q' that is over k blocks $\{R_{V_1}, \cdots, R_{V_k}\}$ as well as the ordering of the blocks in a MNLCP evaluation of Q' where R_{V_1} and R_{V_k} are, respectively, the outermost and innermost relations of the MNLCP evaluation.

With the MNLCP evaluation, for a CPQ $Q' = (R_{V_1}, \cdots, R_{V_k})$, each result tuple $(t_1, t_2, \cdots, t_k)$ of Q' (where each tuple $t_i \in R_{V_i}$) is constructed progressively as a sequence of partial result tuples: (t_1), (t_1, t_2), $\cdots$, and finally $(t_1, t_2, \cdots, t_k)$. To optimize the MNLCP evaluation, for each partial result tuple $t = (t_1, t_2, \cdots, t_j)$ $(1 \leq j < k)$, we check whether t satisfies each anti-monotone set predicate p in P_0. If t does not satisfy p, then this implies that none of the partial result tuples extended from t will satisfy p; therefore, the MNLCP evaluation involving t can be immediately "short-circuited" by dropping t from further processing. Note that similar optimization is also applicable for the monotone set predicates in general SQs. Specifically, if each partial results tuple t satisfies p, then we can conclude that each of the partial result tuple extended from t will also satisfy p. Further optimizations for anti-monotone/monotone set predicates evaluation are discussed in Section 7.2.

The number of CPQs evaluated for a BSQ can be very large: the maximum number of CPQs when n ranges from 3 to 7 are 7, 48, 461, 6432, and 129424, respectively. Therefore, there could be considerable efficiency gains by applying multi-query optimization (MQO) techniques to optimize the evaluation of a BSQ. However, MQO is a very hard optimization problem with a search space that is doubly exponential in the size of the queries [12, 13, 14, 15]. As early exhaustive strategies [12, 14] are not practical, many heuristic solutions have been proposed (e.g. [16, 13, 15, 17, 18, 19]). To cope with the high optimization complexity, a well-known strategy for MQO is to adopt a two-phase optimization approach [19, 18]. The first phase generates local optimal query plans for the individual queries, and the second phase generates a global query plan that exploits the common subexpressions (CSEs) in the local query plans.

However, the existing MQO heuristics are not appropriate for our problem context for two main reasons. First, as explained above, the number of CPQs in our problem is very large, which means that it is important to use an efficient heuristic that can scale to thousands of queries. Existing MQO heuristics are, however, not designed for such scale. As an example, the state-of-the-art MQO heuristic [13] took 30 seconds to optimize 22 (which is the maximum number of queries considered) queries without considering cross product joins where each query only references five relations,

and was unable to scale when the number of relations in the queries increases or cross product joins are considered. Second, most of the existing MQO works [12, 13, 14, 15, 19] are based on the materialization and reusing the results of CSEs which is not beneficial for our context. This is because for CPQs, the cost of computing, writing and reading a CSE result to/from disk is higher than the cost of recomputing the CSE as shown by our experimental results in Section 8.1. Thus, our approach for evaluating CPQs does not employ the materialization technique; instead, we evaluate them by pipelining the results of CSEs to CPQs.

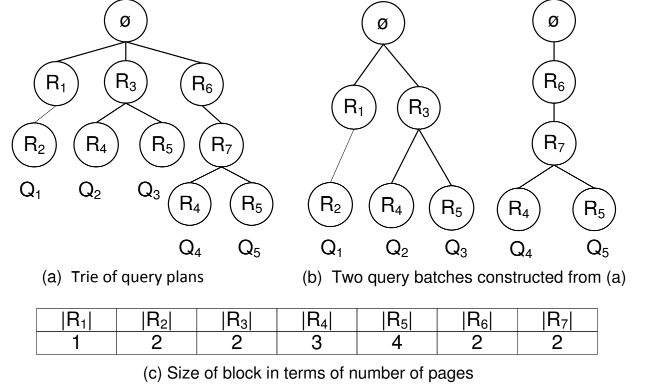
Due to both the scale of the problem as well as the nature of the queries (i.e., CPQs and not join queries), existing MQO heuristics designed for optimizing a moderate number of general join queries are too complex and not sufficiently scalable for our problem. We therefore propose a novel and efficient heuristic, which is also based on the two-phase approach, to optimize the evaluation of a large collection of CPQs. The first phase generates a local optimal evaluation plan for each CPQ and the second phase optimizes the collection of local plans by exploiting CSEs.

In the first phase, since each CPQ is evaluated using the MNLCP method, the local evaluation plan for a CPQ is simply a specification of the ordering of the blocks in the CPQ (i.e., from outermost to innermost relation). To optimize the evaluation of CPQs, it is desirable to minimize the cost to check anti-monotone set predicates (to find short-circuited partial result tuples). Therefore, our approach to order the blocks for a CPQ is to order them in non-decreasing order of their cardinalities. The intuition behind the approach is to minimize the cost to check the short-circuited partial result tuples assuming that any pair of partial result tuples of the same length are equally likely to be short-circuited. As shown by our cost model in Section 6.2.2, our approach to order the blocks for a CPQ in a MNLCP evaluation is indeed optimal.

In the second phase, to efficiently identify the CSEs among the local query plans, our heuristic uses a trie to represent all the local query plans. Each node in the trie, except for the root node which is a virtual node, represents a block, and each path from a child node of the root node to a leaf node corresponds to the sequence of blocks (in non-decreasing order of their cardinalities) in a local query plan. With this simple technique, our heuristic is able to capture the common "prefixes" among the local query plans. The time complexity of constructing the trie is proportional to the total number of blocks in all the CPQs. The simplicity of this structure enables our heuristic to scale to a large number of queries.

Once the trie has been constructed with the local query plans, the global query plan is formed and evaluated by a top-down traversal of the trie structure. Consider a trie node R_i that has multiple child nodes, and let $(R_1, \cdots, R_{i-1})$ be the path of ancestor nodes of R_i in the trie (i.e., R_1 is the child of the root node and each R_j is a child node of R_{j-1}, $j \in [2, i]$). By pipelining the output of $(R_1 \times \cdots \times R_i)$ to each of the child nodes of R_i, the computation of the cross-product expression associated with the common prefix path is shared among the child nodes.

EXAMPLE 5. *Consider a BSQ that is evaluated as five CPQs $\{Q_1, \cdots, Q_5\}$ with their local query plans shown by the trie in Figure 2(a), where the node labeled $\emptyset$ represents the virtual root node of the trie. Each path from a child node of the root node to a leaf node corresponds to a local query plan for a CPQ. For example, the fourth leftmost path corresponds to the local plan (R_6, R_7, R_4) for Q_4. Observe that the two local plans for Q_2 and Q_3 share the block R_3. Thus, for every tuple t read from R_3, the global plan evaluation will pipeline t to its child nodes R_4 and R_5.*



$\lvert R_1 \rvert$	$\lvert R_2 \rvert$	$\lvert R_3 \rvert$	$\lvert R_4 \rvert$	$\lvert R_5 \rvert$	$\lvert R_6 \rvert$	$\lvert R_7 \rvert$
1	2	2	3	4	2	2

(c) Size of block in terms of number of pages

Figure 2: An example of CPQ blocks from a BSQ organized as a trie.

6. HANDLING LARGE DATA

In this section, we extend our in-memory approach discussed in the previous section to evaluate BSQs on large, disk-based data.

In the following discussion, we use B to denote the number of main memory buffer pages available for evaluating a BSQ Q on a relation R. For a block R_{V_i}, we use $|R_{V_i}|$ and $\|R_{V_i}\|$ to respectively denote its size in terms of number of pages and its cardinality in terms of number of tuples. We assume that the answer sets computed for a BSQ are directly output without being buffered.

6.1 Phase 1: Partitioning Phase

In the first phase, we need to allocate the available buffer space for reading R as well as creating the blocks of R. This partitioning problem using limited buffer space can be solved with two standard database techniques (i.e., sorting and hashing), which we briefly described in this section.

In the hash-based approach, we allocate one buffer page for reading R and divide the remaining buffer pages uniformly among the maximum number of $2^n - 2$ blocks to be materialized[4]. Each tuple read from R is copied to the appropriate block buffer, and a block buffer is flushed to disk when it becomes full. For the case where there is not enough buffer space to even allocate one page for each block, then R will have to be partitioned in multiple passes instead of a single pass.

In the sort-based approach, each tuple read from R is assigned an appropriate block identifier (i.e., $1, \cdots, 2^n - 2$) based on the subset of member predicates that it satisfies. The tuples are then sorted on this identifier using external merge-sort algorithm to create the blocks.

If the buffer space is sufficiently large such that R can be hash partitioned in one scan, then the hash-based approach is generally more efficient as the sort-based approach might require multiple merge passes to sort R. However, if a BSQ contains certain type of set predicates, then the sort-based approach could be optimized to become more efficient; we defer the discussion of the optimization to Section 7.2.

6.2 Phase 2: Enumeration Phase

The main challenge in the second phase is how to efficiently evaluate a large collection of CPQs given a buffer space constraint of B pages.

Consider a CPQ $Q' = (R_1, \cdots, R_k)$. What is an optimal ap-

[4]Recall from Section 5 that $R_\emptyset$ and R_T are not materialized.

proach to evaluate Q' such that (1) the buffer space used is minimized and (2) each block in Q' is read only once? A well-known strategy [20, 21] to achieve this is to load all the blocks of Q', except for the outermost block (i.e., R_1), into the buffer and to allocate only one buffer page for R_1. As each page b of R_1 is loaded into the buffer, the MNLCP method is used to compute $b \times R_2 \times \cdots \times R_k$. Thus, the minimum buffer space required for this optimal evaluation is $1 + \sum_{i=2}^{k} |R_i|$ pages. Let $minbuf(Q')$ denote the minimum buffer space requirement (in terms of number of pages) for evaluating a CPQ Q' in this manner.

Given a buffer space of B pages, we classify a CPQ Q' as a *lean query* if $minbuf(Q') \leq B$; otherwise, Q' is classified as a *fat query*. Let Q_{lean} and Q_{fat} denote the set of all the lean and fat CPQs, respectively, from the collection of CPQs to be evaluated. From the optimization viewpoint, Q_{lean} are easier to optimize than Q_{fat}. Therefore, our proposed approach optimizes the evaluation of Q_{lean} and Q_{fat} separately.

6.2.1 Evaluation of lean queries.

To exploit the CSEs among a collection of lean CPQs, we present an efficient strategy to evaluate them in *batches* such that each batch of queries can be evaluated efficiently similar to the in-memory approach described in Section 5 using only B buffer pages. We first formally define a query batch and then present efficient heuristics to optimize both the partitioning of Q_{lean} into query batches as well as the evaluation order of the batches.

Consider a set of lean CPQs $Q_{batch} = \{Q_1, \cdots, Q_m\}$, where $Q_{batch} \subseteq Q_{lean}$ and each $Q_i = (R_{i,1}, \cdots, R_{i,k_i})$. Let $D(Q_{batch}) = \bigcup_{Q_i \in Q_{batch}} \{R_{i,2}, \cdots, R_{i,k_i}\}$ denote the set of distinct blocks in all the m CPQs from Q_{batch} after excluding the outermost block from each CPQ (i.e., $R_{i,1}, i \in [1, m]$). We say that Q_{batch} forms a *query batch* if $1 + \sum_{R_i \in D(Q_{batch})} |R_i| \leq B$. Note that a query batch Q_{batch} can be evaluated optimally using only B buffer pages as each block (involved in Q_{batch}) is read only once from disk.

EXAMPLE 6. *Assume that $B = 10$. Consider Q_5 in Figure 2. Since $minbuf(Q_5) = 1 + |R_7| + |R_5| = 7 < B$, Q_5 is classified as a lean query. Similarly, all the other queries (Q_1 to Q_4) in Figure 2 are also classified as lean queries. Consider a set of lean queries $Q'_{batch} = \{Q_4, Q_5\}$. We have $D(Q'_{batch}) = \{R_7, R_4\} \bigcup \{R_7, R_5\} = \{R_7, R_4, R_5\}$. Since the total size of the blocks in $D(Q'_{batch})$ (i.e., $|R_7| + |R_4| + |R_5| = 9$) is no larger than $B-1$, Q'_{batch} forms a query batch. On the other hand, for the set of lean queries $Q''_{batch} = \{Q_1, Q_4, Q_5\}$, since the total size of the blocks in $D(Q''_{batch}) = \{R_2, R_7, R_4, R_5\}$ is 11 which is larger than $B - 1$, Q''_{batch} is not a query batch.*

Partitioning of query batches. Since a block may appear in multiple CPQs which are in different query batches, a block may still be read into the buffer multiple times. Thus, it is desirable to group CPQs that share some common block (or more generally, share some CSEs in the form of a subset of blocks) in the same query batch to minimize both the number of times a common block is read into the buffer as well as the number of redundant computations of the CSEs.

Our heuristic to partition Q_{lean} into query batches applies the same idea from Section 5 to organize the CPQs in Q_{lean} using a trie to capture the common "prefixes" among the CPQs. The query batches are then created by a pre-order traversal of the trie as follows. We first initialize the current query batch Q_{batch} to be empty. Whenever the pre-order traversal visits a leaf node in the trie, we have found a CPQ Q' which corresponds to the root-to-leaf path in the trie. If Q_{batch} remains a query batch after Q' is added to it, we add Q' to be part of Q_{batch}; otherwise, we initialize a new query batch with only Q' in it and call this the current query batch. At the end of the traversal, Q_{lean} is partitioned into query batches. By partitioning Q_{lean} in this way, our heuristic is able to capture the CSEs among the CPQs in each batch. Thus, each query batch is a trie which is a subtree of the input trie. The time complexity for the query batch partitioning is linear to the number of nodes in the trie.

Evaluation order of query batches. We now explain how a query batch $Q_{batch} = \{Q_1, \cdots, Q_m\}$ formed using the above approach is evaluated similar to the in-memory approach. Here each Q_i represents a CPQ. For each $Q_i = (R_{i,1}, \cdots, R_{i,k_i}) \in Q_{batch}$, we load into the buffer all the blocks of Q_i, except the outermost block $R_{i,1}$. Note that within each query batch, each block is loaded exactly once in the buffer even if the block appears in different queries. By the definition of a query batch, the remaining number of pages left in the buffer (denoted by B') after loading all the blocks except for $R_{i,1}$ must be at least one. Therefore, we can incrementally load the outermost block $R_{i,1}$ for each Q_i into the buffer (B' pages at a time), and pipeline the loaded tuples of $R_{i,1}$ to each child block to compute the CPQs in the query batch.

The final optimization issue to consider is how to order the query batches formed for evaluation. If two query batches have many blocks in common, then it is desirable to evaluate these two batches consecutively so as to minimize the number of times the same block is loaded into the buffer (across query batches). This scheduling optimization problem can be formulated as finding the longest Hamiltonian path in a fully-connected, weighted, undirected graph $G = (V', E')$ as follows. Each vertex in V' represents a query batch, and each edge in E' has a weight that is equal to the sum of the sizes of the common blocks (excluding the outermost block in each CPQ) between the CPQs corresponding to the connected vertices. This optimization problem is in general NP-complete; and we solve this using a simple $3/4$ approximation algorithm [22], which has a time complexity $O(|V'|^3)$ where $|V'|$ is the number of query batches.

EXAMPLE 7. *Assume that $B = 10$. Figure 2(b) shows two query batches, $Q'_{batch} = \{Q_1, Q_2, Q_3\}$ and $Q''_{batch} = \{Q_4, Q_5\}$, constructed from the trie in Figure 2(a) by a pre-order traversal of the trie. Let us assume that Q'_{batch} is evaluated before Q''_{batch}. When evaluating the batch Q'_{batch}, the blocks R_2, R_4 and R_5 are completely loaded into the buffer and the remaining 1 buffer page is used to load in the tuples in R_1 and R_3 sequentially with the tuples being pipelined to the corresponding children blocks. When evaluating the batch Q''_{batch}, as R_4 and R_5 have already been loaded in the buffer, we only need to load in R_7 (i.e., R_2 is evicted from the buffer) and the remaining 1 buffer page is used to load in R_6.*

6.2.2 Evaluation of Fat Queries.

Since each fat CPQ can not be evaluated optimally with the available B buffer space, our evaluation approach for lean CPQs is not applicable for fat CPQs. To exploit the CSEs among a collection of fat CPQs, another alternative strategy is to materialize and reuse (instead of pipelining) the results of CSEs. However, since a cross-product result is always larger than the combined size of its input operands, a materialization strategy incurs a high I/O cost to write and read the materialized results. Indeed, as shown by our experimental results, it is overall more efficient to recompute the results of a CSE (incurring a higher CPU cost) than to materialize and reuse the results of a CSE. Thus, we propose to use the MNLCP method to evaluate each fat CPQ separately without relying on any result materialization. The main challenge here is how to effectively allo-

cate the buffer space among the blocks in the CPQ to optimize both CPU and I/O costs.

In the following, we first analyze the I/O and CPU costs of the MNLCP evaluation method, and then present our heuristic to optimize the buffer allocation based on these cost models.

Cost models. Consider the evaluation of a fat CPQ $Q' = (R_{V_1}, \cdots, R_{V_k})$ using the MNLCP approach. Let $(b_1, \cdots, b_k)$ denote the buffer space allocation for the blocks, where each R_{V_i} is allocated b_i number of buffer pages, such that $\sum_{i=1}^{k} b_i \leq B$. The MNLCP evaluation method will first load the first b_i pages of each R_{V_i} into the buffer and compute the cross-product among the tuples in the buffer, and then load in the next b_1 pages for R_{V_1}, and so on. Whenever all the pages of some R_{V_i} have been read and loaded into the buffer, the method will load in the next b_{i+1} pages for $R_{V_{i+1}}$ and "rewind" each R_{V_j}, $j \in [1, i]$, by loading in the first b_j pages for each R_{V_j}, $j \in [1, i]$. The method terminates when all the pages of R_{V_k} have been read. The I/O cost to evaluate Q' in such a manner is given by

$$C_{i/o} = \sum_{i=1}^{k} c_{i/o} |R_{V_i}| \prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil \quad (1)$$

where $c_{i/o}$ is the cost ratio to read one page. Each $c_{i/o}|R_{V_i}| \prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil$ represents the I/O cost to load in R_{V_i} with $\prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil$ representing the times to load in R_{V_i}. The CPU cost to evaluate Q' is given by

$$C_{cpu} = \sum_{i=1}^{k} c_{cpu} S_i \prod_{j=1}^{i} \|R_{V_j}\| \prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil \quad (2)$$

where c_{cpu} is the cost ratio to process a tuple and S_i is the selective factor of anti-monotone set predicates for $(i-1)$-sets. Each $c_{cpu} S_i \prod_{j=1}^{i} \|R_{V_j}\| \prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil$ represents the CPU cost to compute the cross product of $(R_{V_1}, \cdots, R_{V_i})$ with $S_i \prod_{j=1}^{i} \|R_{V_j}\|$ representing the number of cross product results that need to be computed, and $\prod_{j=i+1}^{k} \lceil \frac{|R_{V_j}|}{b_j} \rceil$ representing the times to compute the cross-product results. Note that both $c_{i/o}$ and c_{cpu} are tunable constants commonly used in query optimizers and S_i can be estimated based on conventional RDBMS estimation techniques (e.g. with histograms).

We remark that if each b_i $(1 \leq i \leq k)$ is allocated $|R_{V_i}|$ pages (i.e., in-memory case), then our approach to evaluate each CPQ with the blocks ordered in non-descending order of cardinality indeed minimizes the CPU cost.

Optimizing buffer allocation. As both $C_{i/o}$ and C_{cpu} are not related to b_1, we will allocate the minimum of one page to b_1. The overall optimization problem is to minimize $C_{total} = C_{i/o} + C_{cpu}$ with the following constraints: (1) $b_1 = 1$, (2) $\sum_{i=2}^{k} b_i \leq B - 1$, and (3) $b_i \leq |R_{V_i}|$ for $2 \leq i \leq k$.

A naive solution to optimize the above is to try all possible assignments for $(b_2, \cdots, b_k)$. However, the time complexity will be $O(B^k)$ which is not feasible when B and k are large. Therefore, we use a simple greedy approach to solve the problem by iteratively selecting the "best" block to increase its buffer allocation until the buffer space is fully utilized. Initially, each block is allocated one page (i.e., $b_i = 1$ for $2 \leq i \leq k$). At each iteration, we first compute the *benefit ratio* for each block R_{V_i}, given by $(C - C'_i)/(b'_i - b_i)$, where b_i is the current buffer allocation for R_{V_i}, C is C_{total} for the current buffer allocation, b'_i is the smallest possible integer such that $b'_i > b_i$ and $\lceil \frac{|R_{V_i}|}{b'_i} \rceil < \lceil \frac{|R_{V_i}|}{b_i} \rceil$, and C'_i is C_{total} after increasing b_i to b'_i. Thus, the benefit ratio measures the reduction in evaluation cost per additional buffer page allocated for a block. Then we increase b_i for the block R_{V_i} with the maximum benefit ratio to b'_i. The time complexity of this heuristic is $O(Bk)$ where B is the maximum number of iterations and $O(k)$ is the time complexity of an iteration.

Unlike lean CPQs, where the order of evaluation is optimized, we do not optimize the order of evaluating fat CPQs as the potential benefit is questionable. Since the allocated buffer for a block is generally less than the block size, and the allocation could vary among CPQs having that block, we can only partially share the scan of the block across CPQs which entails non-trivial bookkeeping to keep track of partially loaded blocks. We therefore do not consider this optimization in this paper.

6.3 Progressive Approaches

Our proposed two-phase approach is a blocking algorithm in that the enumeration phase can only start after the partitioning phase has completed. For a BSQ that does not require retrieving all the answer sets (e.g., the query has a limit-clause), this approach is not ideal. In this section, we describe how to extend the two approaches (sort-based and hash-based approaches) for the first phase to make them non-blocking (i.e., progressive) so that more answer sets can be generated earlier during the first phase (beyond those produced by R_T). The challenge is to avoid generating duplicate answer sets that are produced in both the partitioning and enumeration phases.

Sort-based approach. To make the sort-based partitioning phase progressive, we generate answers while creating initial sorted runs as follows. For each set of tuples that form an initial sorted run, we first sort them based on their block identifiers, and then generate minimal answer sets using these in-memory blocks following the basic approach described in Section 5. In this way, we are able to compute some answer sets as initial sorted runs are being created in the partitioning phase. A simple way to avoid generating duplicate answer sets is to simply assign a run number to each tuple in the partitioning phase and detect for duplicate answer sets during the enumeration phase as follows: if all the tuples in a potential answer set have the same run number, then the set is a duplicate and is ignored.

Hash-based approach. To make the hash-based partitioning phase progressive, we simply generate answer sets for each new tuple t read with all the in-memory tuples (i.e., we construct the trie for the block containing t with all the in-memory blocks). In the event that the buffer space is full, we make room for t by selecting some other in-memory block and flush it to disk. To detect for duplicate answer sets, we adapted the techniques from [23, 24] as follows. Each tuple t is assigned a timestamp [*begin*, *end*], where *begin* and *end* represent, respectively, the time t is read into memory and the time t is flushed to disk. Thus, for each potential answer set S considered in the enumeration phase, S is a duplicate answer if the intersection of the timestamps of all the tuples is not empty.

Comparing the two approaches, the hash-based approach may produce results earlier than the sort-based approach since the former can produce results immediately for each newly read tuple while the latter can only produce results after it has filled and sorted the buffer with tuples. However, the hash-based approach is likely to run slower than the sort-based approach due to the per-tuple overhead (i.e., trie construction for each tuple).

7. EXTENSIONS AND OPTIMIZATIONS

In this section, we first extend our proposed approaches to evaluate SQs in Section 7.1. We then discuss the further optimization

of SQ evaluation for sort-based approaches based on the properties of set predicates in Section 7.2.

7.1 Evaluation of SQs

To evaluate SQs, our proposed approaches for BSQs can be extended as follows.

In the partitioning phase, the input table R is partitioned as before based on the combination of predicates satisfied by the tuples; however, we now need to materialize both blocks $R_\emptyset$ and R_T. This is because for a SQ Q, it is now possible for $S \cup \{t\}$ to be an answer set for Q, where $t \in R_\emptyset \cup R_V$ and S is a set of tuples from the blocks excluding $R_\emptyset$ and R_V. Hence, both $R_\emptyset$ and R_V need to be materialized for generating potential answer sets in the second phase.

Since the answer sets for SQs are not necessarily minimal and the set predicates in SQs are not necessarily anti-monotone, the enumeration phase now requires a weaker definition of vbset (Section 5) that satisfies only property P1. This weaker definition has two implications. First, the blocks in a vbset are now not necessarily distinct as it is possible for an answer set to contain multiple tuples from the same block. However, as the cardinality of answer sets is bounded by n, the maximum number of blocks in a vbset is also bounded by n. Second, it is now possible for one vbset to be a subset of another vbset. For instance, in the example SQ in Section 1, if the query is not constrained to retrieve only minimal answer sets, then both $\{R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}\}$ and $\{R_{\{v_1,v_3\}}, R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}\}$ are vbsets with one being a subset of the other.

Consequently, after constructing the trie to capture the CSEs for the local query plans, each path from a child node of the root node to any node in the trie now may correspond to a vbset. Note that this is different from the trie constructed for BSQs where only a path from a child node of the root node to a leaf node corresponds to a vbset. Furthermore, since a vbset U could contain multiple instances of the same block, the CPQ corresponding to U needs to be evaluated such that answer sets with duplicates are not generated by judicious manipulation of tuple pointers using the MNLCP approach[5].

EXAMPLE 8. *Figure 3 shows the trie to represent the local query plans for a SQ that is evaluated as six CPQs* $\{Q_6, Q_7, Q_8, Q_9, Q_{10}, Q_{11}\}$*, where the node labeled* $\emptyset$ *represents the virtual root and each path from a child node of the root node to a node labeled with* Q_i ($i \in [6, 11]$) *represents a local plan for a CPQ. Note that for a SQ, a CPQ may contain multiple instances of the same blocks (e.g.,* Q_6 *and* Q_8*). Different with the trie constructed for a BSQ where only a path from a child node of the root node to a leaf node corresponds to a vbset, each path from a child node of the root node to any node in the trie constructed for a CPQ now may correspond to a vbset (e.g.,*Q_7*).*

Minimal set constraint. For SQs that are constrained to retrieve only minimal sets, the following additional extensions are required. In the partitioning phase, for R_V, if a tuple t in R_V satisfies P_0, then we simply output t as a singleton answer set; otherwise, we materialize t. Thus, the materialized R_V contains tuples that satisfy all the member predicates but do not satisfy P_0. In the enumeration phase, since the weaker vbset definition does not guarantee that a candidate answer set is minimal, we need to verify its minimality requirement during the enumeration phase as discussed in Section 4.

[5]Consider the evaluation of a CPQ $(R_1, R_2, \cdots)$ where R_1 and R_2 are two instances of the same block R. To avoid generating duplicate answer sets, whenever the tuple pointer for the outer block R_1 is moved to the ith tuple of R, the tuple pointer for the inner block R_2 is rewind to the $(i+1)^{th}$ (rather than the first) tuple of R.

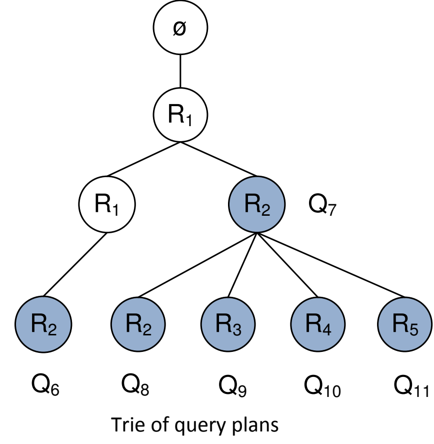


Figure 3: An example of CPQ blocks from a SQ organized as a trie

EXAMPLE 9. *Consider the example SQ* Q_{poi}*. In the partitioning phase,* R *is partitioned into five blocks:* $R_{\{v_1,v_3\}} = \{t_1\}$, $R_{\{v_2,v_4\}} = \{t_2\}$, $R_{\{v_2,v_3\}} = \{t_3\}$, $R_{\{v_1\}} = \{t_4\}$ *and* $R_\emptyset = \{t_5\}$*. In the enumeration phase, all the vbsets are enumerated using the weaker definition of vbset for SQs. Some example vbsets include* $\{R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}\}$, $\{R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}, R_{\{v_2,v_3\}}\}$ *and* $\{R_{\{v_1,v_3\}}, R_{\{v_2,v_4\}}, R_\emptyset\}$*. Note that the number of vbsets for the SQ is larger than that for the corresponding BSQ (Example 3) due to the weaker definition of vbset for SQs. After evaluating the corresponding CPQs and checking the set predicates, three answer sets* $\{t_1, t_2\}$,$\{t_1, t_2, t_3\}$ *and* $\{t_1, t_2, t_5\}$ *are formed.*

7.2 Optimizations of SQ Evaluation

In this section, we describe how the evaluation of a SQ using MNLCP can be further "short-circuited" for sort-based approach by exploiting the presence of certain set predicates in the SQ. Before we describe the optimizations, we first present some preliminaries.

Consider a function F that takes a set of tuples S as input and outputs a numeric value. F is classified as *distributive* if there exists another function F' such that for each pairwise disjoint partitioning of $S = S_1 \cup \cdots \cup S_m$, $F(S) = F'(\{F(S_1), \cdots, F(S_m)\})$. A distributive function F is classified as *monotone* if for any two sets of tuples of the same cardinality, $S_1 = \{t_1, \cdots, t_k\}$ and $S_2 = \{t'_1, \cdots, t'_k\}$, if $F(\{t_i\}) \leq F(\{t'_i\})$ for each $i \in [1, k]$, then $F(S_1) \leq F(S_2)$. The function "SUM(S.duration)" in Section 2 is an example of a distributive monotone function.

Anti-monotone set predicates. If a SQ contains an anti-monotone set predicate p of the form $F(S) \leq c$ where F is a distributive monotone function, then we can optimize the sort-based approach of partitioning as follows. Instead of sorting the tuples using only the block identifier pid, we sort on the composite key $(pid, F(\{t\}))$ which generates blocks that are sorted on $F(\{t\})$. When evaluating a CPQ $(R_{V_1}, \cdots, R_{V_k})$ using MNLCP to generate k-sets, if t_j is the first tuple from R_{V_j} $(1 \leq j \leq k)$ that does not satisfy p when combined with a specific combined tuple $(t_1, \cdots, t_{j-1})$ from $(R_{V_1} \times \cdots \times R_{V_{j-1}})$, then we can short-circuit the MNLCP evaluation by dropping $(t_1, \cdots, t_{j-1})$ from further processing. Note

Table 4: Compared Algorithms

Notation	Algorithm
ps	progressive, sort-based algorithm
ns	non-progressive, sort-based algorithm
ph	progressive, hash-based algorithm
nh	non-progressive, hash-based algorithm
bs	baseline SQL solution

Table 5: Key Experimental Parameters

Parameter	Notation	Default
Cardinality of synthetic input table R	$\|\|R\|\|$	1,000,000
Work memory	B	40MB/200MB
Maximum number of returned answer sets	k	ALL
Number of member predicates	n	4
Selectivity factor of each member predicate	f	0.05
Aggregate value in set predicate	c	Avg

that if we do not sort on the composite key $(pid, F(\{t\}))$, we can only drop $(t_1, \cdots, t_j)$ from processing.

Monotone set predicates. Consider a SQ that contains a monotone set predicate p of the form $F(S) \geq c$ where F is a distributive monotone function, then we can optimize the sort-based approach of partitioning as follows. Here again, we sort on the composite key $(pid, F(\{t\}))$ which generates blocks that are sorted on $F(\{t\})$. When evaluating a CPQ $(R_{V_1}, \cdots, R_{V_k})$ using MNLCP to generate k-sets, if t_j is the first tuple from R_{V_j} $(1 \leq j \leq k)$ that satisfies p when combined with a specific combined tuple $(t_1, \cdots, t_{j-1})$ from $(R_{V_1} \times \cdots \times R_{V_{j-1}})$, then we do not need to check the satisfaction for the partial result tuples extended from $(t_1, \cdots, t_{j-1})$. Note that if we do not sort on the composite key $(pid, F(\{t\}))$, we can only avoid the satisfiability checking for the partial result tuples extend from $(t_1, \cdots, t_j)$.

Due to the fixed cardinality of the answer sets for a vbset, the above optimization can also be applied for some functions that are not distributive monotone. One such example is AVG(S.price) $\leq$ (or $\geq$) c.

8. PERFORMANCE STUDY

In this section, we present an experimental study to compare the performance of our proposed approach against the baseline SQL solution. Our approach was implemented on PostgreSQL 8.4.4, and the experiments were performed on an Intel Dual Core 2.33GHz machine with 3.2GB of RAM and two SATA2 disks running Linux. Both OS and DBMS were installed on a 250GB disk, while the database was stored on a 1TB disk.

Implementation. We implemented our evaluation approach as a new operator inside the PostgreSQL execution engine. An engine-based implementation offers the best performance as it enables the implementation to leverage the existing evaluation code (e.g., external sorting and hashing). Furthermore, it makes the interaction with other database operators much easier. For example, the results of SQs can be pipelined to other database operators like join and set-skyline to perform additional computation.

Algorithms. Table 4 shows the notations for the five algorithms (four variants of proposed approach and one baseline SQL solution) compared in the experiments. For each non-progressive algorithm A $(A \in \{ns, nh\})$, we use A-p and A-e to represent, respectively, its partitioning and enumeration phases. For the SQL solution, we actually experimented with two variants: the first variant used virtual views while the second variant used materialized views. In the experimental results, each running time shown for the SQL solution refers to the timing of the more efficient variant; furthermore, we omit reporting its running time if it exceeds 12 hours.

Datasets. We used both synthetic and real datasets for the experiments. Our real dataset is from the MusicBrainz database [25] which stores music metadata. We created a materialized view by joining several tables from the database as the input relation for our experiments. The schema of the view is *music(mid,mname,duration, language,aname,atype,abegindate,aenddate,bname,battribute,btype, bscript)*, and the detailed information about the attributes can be found in [25]. After removing tuples with non-positive duration attribute value, the size of the materialized view is 1.35GB with 8,507,949 tuples.

Our synthetic dataset was generated based on the schema of the MusicBrainz database [25]. The size of the relation (in the default setting) is 408MB with 1 million tuples. For attributes used for member predicates, their values were generated with a uniform distribution to simplify our control on the selectively factors, while for the attribute (i.e., duration) used for the set predicate (i.e., sum), its values were generated with a Gaussian distribution $(\mu = 300, \sigma = 55)$ to ensure that each query returns a reasonable number of answer sets.

Queries. Our experimental queries aim to find different subsets of music files to meet certain constraints. We tested on both BSQs and SQs for the experiments. Each query has between 2 to 6 member variables with exactly one member predicate for each member variable. All the member predicates are on different attributes. Each BSQ also has an anti-monotone set predicate of the form $sum(S.duration) \leq c$, while each SQ has the same anti-monotone set predicate as well as a monotone set predicate of the form sum $(S.duration) \geq c/2$, where c is some constant value. Each query was run three times and we report their average running time.

Parameter settings. Table 5 shows the key parameters and their default values used in the experiments; the default parameter values were used unless specified otherwise. The k parameter represents the maximum number of required answer sets in the query's limit clause and has a default value of "ALL" to retrieve all answer sets. The c parameter is used to control the selectivity of the set predicates and its default value (denoted by "Avg") refers to the average value of the *duration* attribute, which is 230 seconds for the real dataset and 300 seconds for the synthetic datasets.

The work memory parameter B controls the main memory allocated in PostgreSQL for our algorithms as well as for sorting and storing hash tables. Since we are interested in comparing the disk-based variants of our algorithms, we set $B = 40$MB for BSQs and $B = 200$MB for SQs in the default setting. Note that a larger B value was used for SQs since the evaluation of SQs requires both $R_\emptyset$ and R_V to be materialized in the partitioning phase which significantly increases the total size of the blocks. However, for the baseline SQL solution, we actually used a larger, fixed value of 256MB of work memory (to improve its performance via speeding up the sort-merge and hash joins in the SQL solution), which is much larger than the typical work memory size recommended for PostgreSQL [26]. Thus, our work memory allocation favors the baseline solution.

Summary of results. For queries where all the query results are returned, our algorithms significantly outperform the SQL solution by up to three orders of magnitude and the non-progressive algorithms are at least as fast as the corresponding progressive algorithms. Furthermore, the sort-based algorithms are significantly faster by up to two orders of magnitude than the corresponding hash-based algorithms due to the optimization technique discussed

in Section 7.2 for sort-based algorithms. However, the partitioning phase of nh is slightly faster than the partitioning phase of ns as discussed in Section 6.

For queries where the maximum number of returned answer sets are limited (i.e., with limit-k clause), our experimental results (with k ranging from 10 to 50) show that both the progressive and non-progressive algorithms outperform the baseline solution by up to one order of magnitude and the progressive algorithms are faster than the corresponding non-progressive algorithms. Furthermore, ph is able to produce results earlier than ps as ph can start to produce results immediately for each newly read tuple while ps needs to fill and sort the buffer with tuples before producing any results.

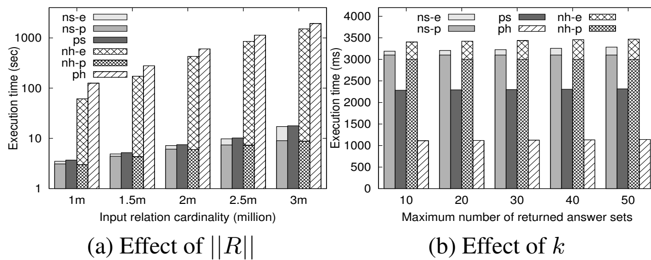
8.1 Results for BSQs on Synthetic Datasets

In this section, we first compare our proposed algorithms against the baseline SQL solution, and then study the effectiveness of our optimizations for evaluating lean and fat CPQs, and finally compare the relative performance of our algorithms for different settings.

8.1.1 Comparison with SQL baseline solution.

Figure 4(a) compares the performance as a function of the input relation cardinality $||R||$. The running times for the baseline solution are not shown on the graph as they are extremely long: for relation cardinality sizes of 1m, 1.5m and 2m, it took 1.2hr, 3.3hr and 6.9hr, respectively; and it exceeded 12hr for cardinality sizes beyond that. Thus, comparing to the cases where the baseline solution run to completion (i.e., under 12hr), our algorithms outperform the baseline solution by up to three orders of magnitude.

As expected, the running times of our algorithms increase with the value of $||R||$. Since a larger input table results in larger blocks, this increases the CPQ processing time for three reasons. First, larger blocks increase the number of results; second, larger blocks cause lean CPQs to be partitioned into more query batches which requires more processing time; and third, larger blocks also increase the number of fat CPQs (which are more costly to evaluate than lean CPQs). For example, when the input cardinality is 1m, 1.5m, 2m, 2.5m, and 3m, the number of answer sets is, respectively, 7942, 15721, 31584, 51247, and 75273; the number of query batches is, respectively, 6, 8, 14, 14, and 15; and the number of fat CPQs is, respectively, 0, 1, 7, 7, and 7.



(a) Effect of $||R||$ (b) Effect of k

Figure 4: Comparison with the baseline solution

To enable the baseline solution to complete running within reasonable time, we also compared the algorithms by limiting the maximum number of returned results by varying the k parameter. The comparison is shown in Figure 4(b).

For the baseline solution, we manually control its running to obtain k results as follows. Recall that the baseline solution works by generating answer sets iteratively (i.e., 1-sets, 2-sets, etc.) using a sequence of queries. We first try to obtain k answer sets from the query that generates answer 1-sets. If k results are obtained, then we are done; otherwise, we try to obtain the remaining answer sets from the query that generates answer 2-sets, and so on until we get k results.

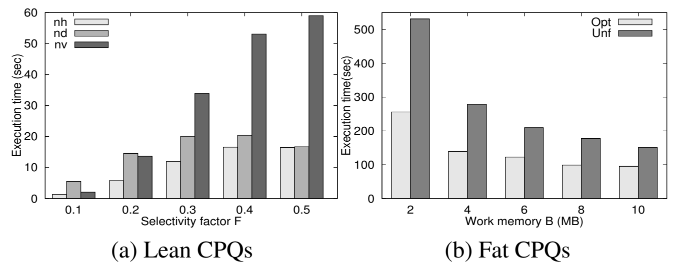
The performance of the baseline solution (results omitted in Figure 4(b)) is almost one order of magnitude slower than our approach: specifically, the running time of bs are 3.6s, 14.9s, 15.0s, 18.1s and 26.9s, respectively, for a k value of 10, 20, 30, 40 and 50. As expected, the execution time of our approach increases as k increases.

8.1.2 Effectiveness of optimizations.

We now study the effectiveness of our optimizations for evaluating lean and fat CPQs.

Lean CPQs. To evaluate the effectiveness of our MQO heuristic (denoted by nh[6]) to process lean CPQs, we created two alternative heuristics to compare against nh. The first heuristic (denoted by nd) is equivalent to nh except for nd uses a different strategy to generate the local plans: for each CPQ, its blocks are ordered in non-increasing order of their cardinalities (i.e., opposite to nh's strategy) for the MNLCP evaluation. nd is used to demonstrate the effectiveness of our heuristic to generate local plans. The second heuristic (denoted by nv) uses the same way as nh to generate local plans. However, unlike nh, nv evaluates the CPQs one at a time without sharing the computations of any CSEs; i.e., nv enumerates the vbsets one by one and process the corresponding CPQs one by one. To enable block scans to be shared, nv employs the following simple buffer replacement strategy: if the buffer is full when a block P is to be loaded into the buffer, nv randomly evicts some block(s) that are not needed by the CPQ being evaluated from the buffer to make room for P. nv is used to demonstrate the effectiveness of our heuristic to share computation of CSEs.

Figure 5(a) compares the running time of nh, nd and nv as a function of selectivity factor of member predicates, f [7]. Note that when f increases from 0.1 to 0.5, the cardinalities of the blocks become more balanced. In particular, when $f = 0.5$, the cardinality of all the blocks are almost the same and thus the running times of nh and nd do not show much differences. The experimental results show that nh outperforms nd by 1.1 times on average and up to 3.2 times when $f = 0.1$, which demonstrates the effectiveness of our MQO heuristic to generate local plans, and nh outperforms nv by 1.7 times on average and up to 2.6 times when $f = 0.5$, which demonstrates the effectiveness of our MQO heuristic to share the computation of CSEs.



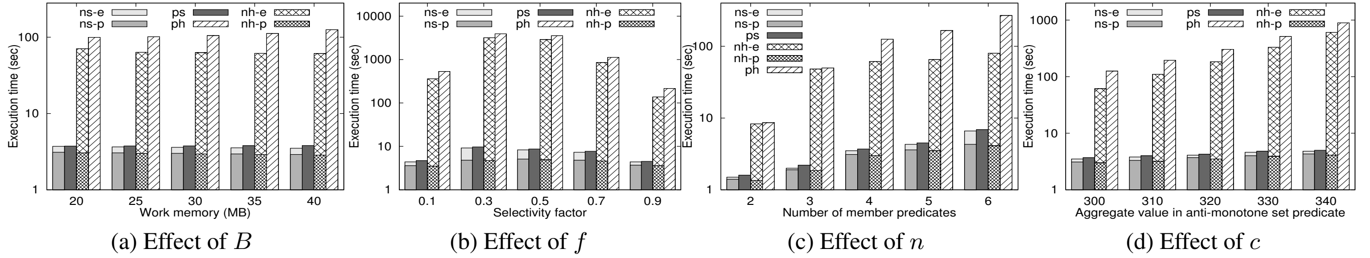
(a) Lean CPQs (b) Fat CPQs

Figure 5: Effectiveness of CPQ optimizations

Fat CPQs. To evaluate the effectiveness of our heuristic technique

[6]We use nh to represent our algorithm since nh is more general than ns (i.e., the optimization technique discussed in 7.2 for ns is only applicable for certain set predicates).

[7]To ensure that all the CPQs in the experiment are lean queries when we vary f, we set $||R|| = 10k$ and $n = 6$.



(a) Effect of B (b) Effect of f (c) Effect of n (d) Effect of c

Figure 6: Effect of varying parameters on synthetic datasets

(denoted by Opt) for processing fat CPQs, we compare against two other competing techniques (denoted by Mat and Unf). The first, Mat, is the materialization strategy discussed in Section 6.2 where a fat CPQ is evaluated as a sequence of binary cross-products with each intermediate result being materialized. The second, Unf, adopts the same MNLCP technique as our Opt but uses a simple buffer allocation strategy that allocates the buffer space uniformly among the query blocks.

To compare the performance of these methods, we created a single fat CPQ with one anti-monotone set predicate that consists of four blocks whose sizes (cardinalities) are, respectively, 3.7MB (7480 tuples), 5.0MB (10084 tuples), 5.1MB (10174 tuples) and 6.3MB (12594 tuples).

Our experimental results show that both Opt and Unf significantly outperform Mat by up to one order of magnitude. As an example, when the work memory is 10MB, the running times for Opt, Unf and Mat are 95s, 151s and 3163s, respectively. Given the poor performance of Mat, we next focus on comparing Opt and Unf as a function of the work memory (i.e., B) in Figure 5(b). As expected, when B increases, the running times for both Opt and Unf decrease. The experimental results show that Opt outperforms Unf by 83% on average and up to 108% when $B = 10$MB.

8.1.3 Effect of Other Parameters.

We compare the effect of other parameters in Figure 6; as before, the results for the baseline solution are omitted here as our algorithms outperform the baseline solution by up to three orders of magnitude.

Figure 6(a) compares the effect of the work memory size, B. As B increases, the running times for the non-progressive algorithms decrease. This is expected since for non-progressive algorithms, the running times for both the partitioning and enumeration phases decrease when B increases. However, the running times for the progressive algorithms increase with more work memory. The reason is that although a larger B speeds up the enumeration phase of the progressive algorithms, it also increases the running time for the partitioning phase of the progressive algorithms since the larger work memory means that more results are produced during the partitioning phase due to the larger buffer of tuples. For the progressive algorithms, our experimental results show that as B increases, the improvement in the enumeration phase is offset by the slower partitioning phase resulting in an overall slower running time.

Figure 6(b) compares the effect of selectivity factor of member predicates, f. We observe an interesting trend where the running time initially increases with increasing f until a certain threshold ($f = 0.3$) after which the running time decreases with increasing f. This is because for BSQs, the value of f affects the type of resultant CPQs and hence the evaluation cost. At one extreme with very small values of f, a tuple is more likely to belong to a block that satisfies a small number of member predicates. Thus, many tuples will belong to the block $R_{\emptyset}$ which means that the resultant CPQs can be evaluated efficiently. At the other extreme with very large values of f, a tuple is more likely to belong to a block that satisfies a large number of member predicates. Thus, the resultant CPQs correspond to vbsets with small cardinality (i.e., CPQs with small number of operand blocks) which can also be evaluated efficiently.

Figure 6(c) compares the effect of the number of member predicates, n. Note that the number of blocks increases exponentially with n. Although a larger number of blocks reduces the number of tuples in each block, it also increases the number of CPQs which increases the running time as shown by our experimental results.

Figure 6(d) compares the effect of selectivity of the set predicate as we increase the aggregate value c in the set predicate. As the value of c increases, the running times for all the algorithms increase. This is expected since the number of results increases (e.g., the number of answer sets are, respectively, 7942, 14905, 27692, 51243, and 94326 for an aggregate value of 300, 310, 320, 330, and 340) with increasing c value which therefore increases the running time.

8.2 Results for BSQs on Real Dataset

In this section, we evaluate the performance of BSQs using the real dataset. Since the cardinality of the real dataset is larger than that of the synthetic datasets, we used smaller selectivity factors for the member predicates for the experiments on the real dataset. In the default setting, each query has four member predicates with the following selectivity factors: 6.1×10^{-4}, 1.1×10^{-3}, 9.4×10^{-4} and 5.8×10^{-4} [8]. Accordingly, we used a smaller default work memory size of 1MB to ensure that we are comparing the disk-based variants of the algorithms.

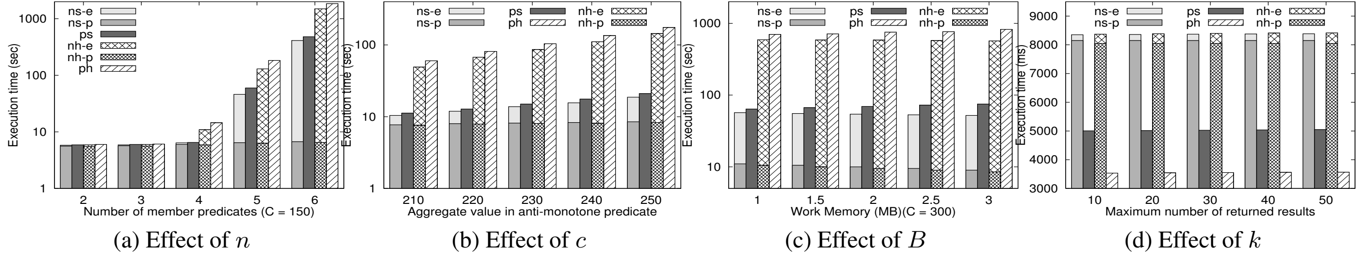
In the default setting, the baseline solution did not complete running in 12 hours. In contrast, the running times of ns, ps, nh and ph are $13.8s$, $15.0s$, $86.4s$ and $104s$ respectively. The results show that our algorithms are at least three orders of magnitude faster than the SQL solution.

Figure 7 compares the effect of varying various parameters using the real dataset. Our experimental results for the real dataset exhibit similar trends observed for the synthetic datasets, and we therefore do not repeat the analysis of the results. In figure 7(d), the running times of the baseline solution are not shown as they are one order of magnitude slower than our algorithms. For example, when $k = 10$, the running times of ph, ps, ns, nh and bs are respectively 3.5s, 5.0s, 8.4s, 8.4s and 84s.

8.3 Results for SQs on Synthetic Datasets

In this section, we evaluate the performance of SQs on synthetic datasets. Our experimental results for SQs show that both SQs and

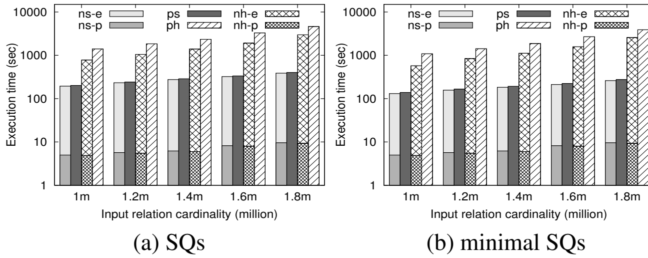
[8] In Figure 7(a), the selectivity factors of the additional two member predicates are 3.6×10^{-4} and 2.3×10^{-4}.



(a) Effect of n (b) Effect of c (c) Effect of B (d) Effect of k

Figure 7: Effect of varying parameters on real dataset

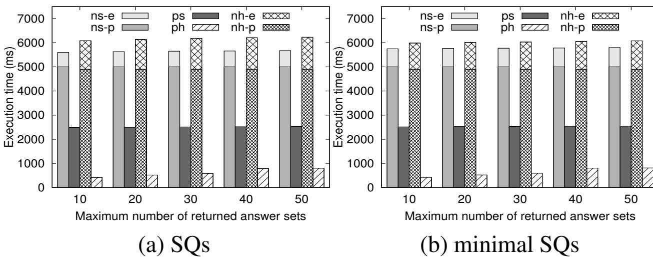
minimal SQs (i.e., SQs that are constrained to retrieve only minimal answer sets) are more time consuming to evaluate than BSQs. For example, in the default setting, the running times of ph for BSQs, minimal SQs and SQs are, respectively, 61s, 575s and 779s. The reason for this is threefold. First, SQs produce more blocks as both $R_\emptyset$ and R_V have to be materialized in the partitioning phase. Second, SQs require more vbsets to be enumerated (due to the weaker definition of vbsets). For example, when $n = 4$, the number of vbsets for BSQs and SQs are, respectively, 48 and 3229. Third, the number of returned answer sets for SQs are larger. For example, the number of answer sets for BSQs, minimal SQs and SQs are, respectively, 7942, 9214 and 15563 (in the default setting). We also observe that minimal SQs can be evaluated more efficiently than SQs as minimal SQs can prune the cross product space for SQs (i.e., if S is a minimal answer set, then all the supersets of S can be pruned).



(a) SQs (b) minimal SQs

Figure 8: Effect of $||R||$

Figure 8 compares the effect of $||R||$ for both SQs and minimal SQs. The baseline SQL solution did not complete execution within 12 hours and we therefore omit its results in the graphs. As expected, the running times of our algorithms increase with $||R||$ as explained in Section 8.1.



(a) SQs (b) minimal SQs

Figure 9: Effect of k

Figure 9 compares the effect of k for both SQs and minimal SQs. As the baseline solution is two orders of magnitude slower than our algorithms, its running times are not shown in the figure. For example, when $k = 50$, the running times of bs are respectively 651.0s and 649.5s for SQs and minimal SQs. As expected, the running times of our algorithms increase slowly with the increasing of k.

We observe that the performance trends for SQs are similar to those for BSQs. Therefore, we do not repeatedly report and discuss them further.

8.4 Results for SQs on Real Dataset

In this section, we evaluate the performance of SQs on the real dataset [9]. Here again, the experimental results show that our algorithms significantly outperform the baseline solution. For example, for SQs in the default setting, the running times of ns, ps, nh and ph are respectively, 0.8hr, 1.33hr, 2.25hr and 5.77hr while the baseline solution did not finish running in 12 hours. Furthermore, even for the setting where only k results are returned, both SQs and minimal SQs for the baseline solution did not finish running in 12 hours. This is because the answer sets for queries on the real dataset have large cardinality due to the low selectivity factors of member predicates as discussed in Section 8.2, the baseline solution has to spend more time to generate large size candidate answer sets before producing any answer sets. Therefore, the baseline solution runs slowly even for limit-k queries.

We do not repeatedly discuss the results for SQs on the real dataset as the trends are similar to the results for SQs on the synthetic datasets.

9. RELATED WORK

There are two main areas related to our work: set-based queries and multi-query optimization.

Set-based queries. Set-based queries aim to find sets of entities of interest to meet certain constraints. There are several works on evaluation of set-based queries: OPAC queries for business optimization problems [5], composite items construction in online shopping applications [6], composite recommendations in recommender systems [8, 9], team formation in social networks [10], set-based preference queries [3] and set-based queries with aggregation constraints [7]. However, the focus of all these works is on optimization SQs whereas our focus is on enumerative SQs. Moreover, as most of these works deal with NP-hard optimization problems, their algorithms are mostly approximate or produce incomplete solutions; in contrast, our algorithm is exact and complete. Finally, our work is focused on optimizing query evaluation at the database engine level, whereas these works is focused on middleware-level solution with mostly main-memory resident data. In summary, our

[9]To compare the disk-based algorithms and reduce the number of answer sets, we set $c = 100$.

work is the first to address efficient techniques to evaluate enumerative SQs.

Multi-query optimization (MQO). MQO aims to find evaluation plans that share CSEs. Most of existing works [27, 28, 16, 29, 30, 12, 13, 14, 19, 15] focus on materializing and reusing the results of CSEs. One of the earliest works [27] uses an iteratively greedy heuristic to select CSEs to materialize given an initial plan produced by a normal query optimizer. The works in [12, 14] describe exhaustive search algorithms and heuristic search pruning techniques to find a global query plan by searching all the plan space. However, the exhaustive search of the plan space incurs high optimization overhead which make these works impractical. The state-of-the-art work in MQO [13] uses AND/OR graphs to capture CSEs and proposes several cost-based greedy heuristics to find a global query plan. However, all these work are not useful for our context since materializing cross-product results is extremely costly and it is less efficient than our proposed non-materialized approach as we have explained and demonstrated.

There are several works [17, 18] that exploit pipelining for MQO. [18] considers specialized MQO techniques to pipeline the results of CSEs for OLAP queries. Their work addresses star join queries where all the dimension tables are assumed to be main-memory resident (i.e., only the fact table is disk-based). In contrast, our MQO techniques are proposed for cross-product queries without any strong assumption about the main-memory residency of the relations.

[17] addresses the MQO problem with pipelining and follows a two-phase optimization strategy which is different from our work. The first phase uses existing techniques (such as [13]) to generate a global plan for a set of queries which is represented as a plan-DAG. All the CSEs that can benefit from materialization are captured by the plan-DAG. The second phase optimizes the plan-DAG by pipelining the results of some CSEs in the plan-DAG. Thus, only the results of CSEs that can benefit from materialization are considered for pipelining. This simplification is restrictive since the results of a CSE could be pipelined to improve performance even if materializing the results of that CSE does not improve performance. Since our work does not materialize the results of any CSEs, their work is not applicable for our context. Furthermore, their work assumes that the pipelined relations/results are not buffered whereas our work focus on efficiently optimizing the buffer allocation for pipelining.

10. CONCLUSION

In this paper, we have proposed a novel and efficient approach to evaluate enumerative set-based queries. Our extensive experimental results demonstrate that our proposed approach significantly outperforms the conventional RDBMS approach by up to three orders of magnitude.

As part of ongoing work, we are examining the evaluation of top-k SQs. In particular, if the ranking function F is a distributive monotone function, then the sort-based evaluation can be optimized as follows. In the partitioning phase, we generate blocks that are sorted on $F(\{t\})$ by sorting the input relation on the composite key $(pid, F(\{t\}))$ where pid is the assigned block identifier. In the enumeration phase, we apply existing rank join algorithms [31] to incrementally produce the ranked answer sets for each vbset and apply the well-known TA algorithm [32] to retrieve the top-k answer sets for all the vbsets.

Acknowledgements This research is supported in part by NUS Grant R-252-000-512-112.

11. REFERENCES

[1] M. Desjardins, K. L. Wagstaff, Dd-pref: A language for expressing preferences over sets, AAAI 2005, pp. 620–626.

[2] M. Binshtok, R. I. Brafman, S. E. Shimony, A. Martin, C. Boutilier, Computing optimal subsets, AAAI 2007, pp. 1231–1236.

[3] X. Zhang, J. Chomicki, Preference queries over sets, ICDE 2011, pp. 1019–1030.

[4] S. Börzsönyi, D. Kossmann, K. Stocker, The skyline operator, ICDE 2001, pp. 421–430.

[5] S. Guha, D. Srivastava, Efficient approximation of optimization queries under parametric aggregation constraints, VLDB 2003, pp. 778–789.

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

[7] Q. T. Tran, C.-Y. Chan, G. Wang, Evaluation of set-based queries with aggregation constraints, CIKM 2011, pp. 1495–1504.

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

[9] M. Xie, L. V. S. Lakshmanan, P. T. Wood, Composite recommendations: from items to packages, Frontiers of computer science, 6 (3) (2012) 264–277.

[10] T. Lappas, K. Liu, E. Terzi, Finding a team of experts in social networks, KDD 2009, pp. 467–476.

[11] M. A. W. Houtsma, A. N. Swami, Set-oriented mining for association rules in relational databases, ICDE 1995, pp. 25–33.

[12] J. Park, A. Segev, Using common subexpressions to optimize multiple queries, ICDE 1988, pp. 311–319.

[13] P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe, Efficient and extensible algorithms for multi query optimization, SIGMOD Rec. 29 (2) (2000) 249–260.

[14] T. K. Sellis, Multiple-query optimization, ACM Trans. Database Syst. 13 (1) (1988) 23–52.

[15] J. Zhou, P.-A. Larson, J.-C. Freytag, W. Lehner, Efficient exploitation of similar subexpressions for query processing, SIGMOD 2007, pp. 533–544.

[16] F.-C. F. Chen, M. H. Dunham, Common subexpression processing in multiple-query processing, IEEE TKDE 10(3), 1998, 493–499.

[17] N. N. Dalvi, S. K. Sanghai, P. Roy, S. Sudarshan, Pipelining in multi-query optimization, J. Comput. Syst. Sci. 66 (4) (2003) 728–762.

[18] Y. Zhao, P. M. Deshpande, J. F. Naughton, A. Shukla, Simultaneous optimization and evaluation of multiple dimensional queries, SIGMOD 1998, pp. 271–282.

[19] K.-L. Tan, H. Lu, Workload scheduling for multiple query processing, Inf. Process. Lett. 55 (5) (1995) 251–257.

[20] W. Kim, A new way to compute the product and join of relations, SIGMOD 1980, pp. 179–187.

[21] J. L. Wolf, B. R. Iyer, K. R. Pattipati, J. Turek, Optimal buffer partitioning for the nested block join algorithm, ICDE 1991, pp. 510–519.

[22] A. Barvinok, E. K. Gimadi, A. I. Serdykov, The maximum TSP, G. Gutin, A. P. Punnen (Eds.), The Traveling Salesman Problem and Its Variations, Vol. 12 of Combinatorial Optimization, Springer, 2007, Ch. 12.

[23] T. Urhan, M. J. Franklin, Xjoin: A reactively-scheduled

pipelined join operator, IEEE Data Eng. Bull 23 (2) (2000) 27–33.
[24] S. D. Viglas, J. F. Naughton, J. Burger, Maximizing the output rate of multi-way join queries over streaming information sources, VLDB 2003, pp. 285–296.
[25] http://musicbrainz.org/doc/musicbrainz_database.
[26] http://www.linux.com/learn/tutorials/394523-configuring-postgresql-for-pretty-good-performance.
[27] P. A. V. Hall, Optimization of single expressions in a relational data base system, IBM Journal of Research and Development, 20 (3), 1976, 244–257.
[28] J. Grant, J. Minker, On optimizing the evaluation of a set of expressions, International Journal of Parallel Programming, 11 (3), 1982, 179–191.
[29] U. S. Chakravarthy, J. Minker, Multiple query processing in deductive databases using query graphs, VLDB, 1986, pp. 384–391.
[30] T. K. Sellis, Global query optimization, SIGMOD 1986, pp. 191–205.
[31] I. F. Ilyas, G. Beskales, M. A. Soliman, A survey of top-k query processing techniques in relational database systems, ACM Computing Survey, 40 (4), 2008, 11:1–11:58.
[32] R. Fagin, A. Lotem, M. Naor, Optimal aggregation algorithms for middleware, PODS 2001, pp. 102–113.

APPENDIX

A. DETAILED ILLUSTRATION OF BASELINE SQL SOLUTION

In this section, we illustrate the baseline solution for evaluating SQs using our example SQ Q_{poi} and BSQs using the BSQ Q'_{poi} that is derived from the SQ Q_{poi} by removing its non-anti-monotone set predicate (i.e., $SUM(S.duration) \geq 6$).

Baseline solution to evaluate the SQ Q_{poi}. Figure 10 shows the SQL queries to evaluate our example SQ Q_{poi}. To simplify the predicates as well as the minimality checking, we create C_1 to represent the information of POIs that satisfy the anti-monotone set predicate (i.e., $SUM(S.duration) \leq 10$). Each tuple in C_1 represents the information for a POI. Each of the four binary valued attributes p_i $(1 \leq i \leq 4)$ indicates whether a POI satisfies P_i, where a value of 1 indicates that the POI satisfies P_i. Note that in Figure 10, to simplify the expression of SQL queries, in the select-clause, $C_i.*$ represents that we retrieve all the attributes in C_i and $C_i.j*$ represents that we retrieve all the attributes from the jth tuple in C_i.

Baseline solution to evaluate the BSQ Q'_{poi}. Recall that there are two SQL-based approaches to evaluate BSQs. Figure 11 shows the SQL queries to evaluate the BSQ Q'_{poi} that generate answer sets in multiple output tables. In Figure 11, we use B_i to denote $C_i \setminus A_i$. Note that for BSQ, C_{i+1} is derived from B_i instead of C_i. In the view A_3, the first four conditions ensure that each answer set in A_3 satisfies all the predicates in Q'_{poi} and the remaining conditions ensure that each answer set in A_3 is minimal, i.e., for each member in the answer set, there must exist some P_i $(1 \leq i \leq 4)$ that is satisfied by only this member in the answer set.

Figure 12 shows the SQL queries to evaluate the BSQ Q'_{poi} that generate all the answer sets in a single output table whose arity is equal to the maximum cardinality of the answer sets given by n. To avoid clutter, we only keep the key attribute *id*. In this approach, since a tuple may satisfy multiple member predicates, the same tuple may appear multiple times (under different columns) within a

```
create view C1(id,duration,p1,p2,p3,p4) as select  id, duration,
case city = S.H. then 1 else 0 as p1, case city = S.Z. then 1 else 0 as p2,
case type = museum then 1 else 0 as p3, case type = park  then 1 else 0 as p4
from R where duration <= 10

create view A1 as select * from C1
where p1 = 1 and p2 = 1 and p3 = 1 and p4 = 1 and duration >= 6

create view C2(id1,duration1,p11,p12,p13,p14,id2,duration2,p21,p22,p23,p24)
as select * from C1 C11, C1 C12
where C11.id < C12.id and C11.duration + C12.duration <= 10

create view A2 as select * from C2 where p11 + p21 > 0 and p12 + p22 > 0
and p13 + p23 > 0 and p14 + p24 > 0 and duration1 + duration 2 >= 6

create view C3(id1,duration1,p11,p12,p13,p14,id2,duration2,p21,p22,p23,p24,
id3,duration3,p31,p32,p33,p34) as select   C21.*, C22.id2* from C2 C21, C2 C22
where  C21.id1 = C22.id1 and C21.id2 < C22.id2 and
C21.duration1 +  C21.duration2 + C22.duration2  <= 10

create view A3 as select * from C3 where p11 + p21 + p31 > 0 and
p12 + p22 + p32 > 0  and p13 + p23 + p33 > 0  and p14 + p24 + p34 > 0
and  duration1 + duration 2 + duration3 >= 6

create viewC4(id1,duration1,p11,p12,p13,p14,id2,duration2,p21,p22,p23,p24,
id3,duration3,p31,p32,p33,p34,id4,duration4,p41,p42,p43,p44) as
select  C31.*, C32.id3* from C3 C31, C3 C32
where  C31.id1 = C32.id1 and C31.id2 = C32.id2 and C31.id3 < C32.id3 and
C31.duration1 + C31.duration2 + C31.duration3 + C32.duration3 <= 10

create view A4 as select * from C4 where p11 + p21 + p31 + p41 > 0  and
p12 + p22 + p32 + p42 > 0 and  p13 + p23 + p33 + p43 > 0 and  p14 + p24 +
p34 + p44 > 0 and  duration1 + duration 2 + duration3 + duration4 >= 6
```

Figure 10: SQL queries to evaluate our example SQ Q_{poi}

row in the result table representing an answer set. Therefore, this approach uses SQL's case statements to check whether a candidate answer set satisfies a set predicate. All the tuples in the view M satisfy all P_i $(0 \leq i \leq 4)$. The view M' removes the answer sets in M that are not minimal. In the view M', the first four conditions ensure that all the members in the m2 tuple are contained in the m1 tuple, and the remaining four conditions ensure that at least one member from the m1 tuple is different from the m2 tuple which guarantees that the m2 tuple is a proper subset of the m1 tuple. The view M'' removes duplicates in M' and stores the answer sets.

B. PROOF OF PROPOSITION 1

PROOF. We prove each of the three properties by contradiction.

Suppose the first property is false; i.e., there exists a block $R_{V_i} \in U$ such that the cardinality of V_i is greater than $n - k + 1$. It follows that U is not a vpset since it does not satisfy the second property of a vpset (i.e., U is not minimal). The reason for this is as follows. To ensure that U is minimal, for any $R_{V_j} \in U$, V_j should contain at least one member variable that other blocks do not contain. Since the cardinality of V_i is greater than $n - k + 1$, the remaining number of member variables is fewer than $k - 1$ which can not ensure that the remaining $k - 1$ blocks in $U \setminus R_{V_i}$ have at least one member variable that other blocks do not contain. Thus, we have a contradiction.

Suppose the second property is false; i.e., for any $R_{V_i} \in U$, the cardinality of V_i is less than $\lceil \frac{n}{k} \rceil$. It follows that U is not a vpset since the number of member variables in $\bigcup_{R_{V_i} \in U} V_i$ is less than n which contradicts the first property.

```
create view C1(id,duration,p1,p2,p3,p4) as select id, duration,
case city = S.H. then 1 else 0 as p1, case city = S.Z. then 1 else 0 as p2,
case type = museum then 1 else 0 as p3, case type = park then 1 else 0 as p4
from R where duration <= 10 and p1 + p2 + p3 + p4 > 0

create view A1 as select * from C1 where p1 = 1 and p2 = 1 and p3 = 1 and p4 = 1

create view B1 as select * from C1 except select * from A1

create view C2(id1,duration1,p11,p12,p13,p14,id2,duration2,p21,p22,p23,p24) as
select * from B1 B11, B1 B12 where B11.id < B12.id and (B11.duration + B12.duration) <= 10

create view A2 as select * from C2 where p11 + p21 > 0 and p12 + p22 > 0
and p13 + p23 > 0  and p14 + p24 > 0

create view B2 as select * from C2 except select * from A2

create view C3(id1,duration1,p11,p12,p13,p14,,id2,duration2,p21,p22,p23,p24, id3,duration3,
p31,p32,p33,p34) as select   B21.*, B22.id2* from B2 B21, B2 B22  where  B21.id1 = B22.id1 and
B21.id2 < B22.id2 and (B21.duration1 + B21.duration2 + B22.duration2 ) <= 10

create view A3 as select * from C3  where p11 + p21 + p31 > 0 and p12 + p22 + p32 > 0 and
p13 + p23 + p33 > 0 and p14 + p24 + p34 > 0 and  ((p11 = 1 and p21 + p31 = 0) or (p12 = 1
and p22 + p32 = 0) or  (p13 = 1 andp23 + p33 = 0) or (p14 = 1 and p24 + p34 = 0)) and  (
(p21 = 1 and p11 + p31 = 0) or (p22 = 1 and p12 + p32 = 0) or (p23 = 1 and p13 + p33 = 0)
or (p24 = 1 and p14 + p34 = 0) ) and  ((p31 = 1 and p11 + p21 = 0) or (p32 = 1
and p12 + p22 = 0) or (p33 = 1 and p13 + p23 = 0) or  (p34 = 1 and p14 + p24 = 0))

create view B3 as select * from C3 except select * from A3

create view C4(id1,duration1,p11,p12,p13,p14,id2,duration2,p21,p22,p23,p24,d3,duration3,
p31,p32,p33,p34,id4,duration4,p41,p42,p43,p44) as select B31.*, B32.id3* from B3 B31, B3 B32
where  B31.id1 = B32.id1 and B31.id2 = B32.id2 and B31.id3 < B32.id3 and
(B31.duration1 + B31.duration2  + B31.duration3 +B32.duration3 ) <= 10

create view A4 as select * from C4 where p11 + p21 + p31 + p41 = 1 and
p12 + p22 + p32 + p42 = 1 and p13 + p23 + p33 + p43 = 1 and  p14 + p24 + p34 + p44 = 1
```

Figure 11: SQL queries to evaluate the BSQ Q'_{poi} that generate results in multiple output tables

Suppose the third property is false; i.e., there exists a pair of distinct blocks R_{V_i} and R_{V_j} in U such that $V_i \subseteq V_j$. It follows that U is not a k-vpset since the subset $U \setminus R_{V_i}$ can also satisfy all the member predicates which contradicts the second property. $\square$

```
create  view M as (id1, id2, id3, id4)
select  R1.id, R2.id, R3.id, R4.id from R R1, R R2, R R3, R R4
where R1.city =  S.H. and R2.city = S.Z. and  R3.type = museum and R4.type = park
and    (R1.duration + case (R2.id = R1.id) then 0 else R2.duration  + case (R3.id = R1.id
        or R3.id = R2.id)  then 0 else R3.duration  + case (R4.id = R1.id or R4.id = R2.id
        or R4.id = R3.id) then 0 else  R4.duration) <= 10

create view M' as select * from M m1 where Not Exists
select * from M m2 where
(m2.id1 = m1.id1 or m2.id1 = m1.id2 or m2.id1 = m1.id3 or m2.id1 = m1.id4) and
(m2.id2 = m1.id1 or m2.id2 = m1.id2 or m2.id2 = m1.id3 or m2.id2 = m1.id4) and
(m2.id3 = m1.id1 or m2.id3 = m1.id2 or m2.id3 = m1.id3 or m2.id3 = m1.id4) and
(m2.id4 = m1.id1 or m2.id4 = m1.id2 or m2.id4 = m1.id3 or m2.id4 = m1.id4) and (
(m1.id1 ≠ m2.id1 and m1.id1 ≠ m2.id2 and m1.id1 ≠ m2.id3 and m1.id1 ≠ m2.id4) or
(m1.id2 ≠ m2.id1 and m1.id2 ≠ m2.id2 and m1.id2 ≠ m2.id3 and m1.id2 ≠ m2.id4) or
(m1.id3 ≠ m2.id1 and m1.id3 ≠ m2.id2 and m1.id3 ≠ m2.id3 and m1.id3 ≠ m2.id4) or
(m1.id4 ≠ m2.id1 and m1.id4 ≠ m2.id2 and m1.id4 ≠ m2.id3 and m1.id4 ≠ m2.id4) )

create view M'' as select * from M' m1 where Not Exist
select * from M' m2 where
(m2.id1 = m1.id1 or m2.id1 = m1.id2 or m2.id1 = m1.id3 or m2.id1 = m1.id4) and
(m2.id2 = m1.id1 or m2.id2 = m1.id2 or m2.id2 = m1.id3 or m2.id2 = m1.id4) and
(m2.id3 = m1.id1 or m2.id3 = m1.id2 or m2.id3 = m1.id3 or m2.id3 = m1.id4) and
(m2.id4 = m1.id1 or m2.id4 = m1.id2 or m2.id4 = m1.id3 or m2.id4 = m1.id4) and
(m2.id1 ≠ m1.id1 or m2.id2 ≠ m1.id2 or m2.id3 ≠ m1.id3 or m2.id4 ≠ m1.id4) and (
(m2.id1 < m1.id1) or (m2.id1 = m1.id1 and m2.id2 < m1.id2) or
(m2.id1 = m1.id1 and m2.id2 = m1.id2 and m2.id3 < m1.id3) or
(m2.id1 = m1.id1 and m2.id2 = m1.id2 and m2.id3 = m1.id3 and  m2.id4 < m1.id4))
```

Figure 12: SQL queries to evaluate the BSQ Q'_{poi} that generate results in a single output table