Scalable Package Queries in Relational Database Systems

Matteo Brucato[UM] Juan Felipe Beltran[†] Azza Abouzied[†] Alexandra Meliou[UM]

[UM]College of Information and Computer Sciences
University of Massachusetts
Amherst, MA, USA
{matteo,ameli}@cs.umass.edu

[†]Computer Science
New York University
Abu Dhabi, UAE
{juanfelipe,azza}@nyu.edu

ABSTRACT

Traditional database queries follow a simple model: they define constraints that each tuple in the result must satisfy. This model is computationally efficient, as the database system can evaluate the query conditions on each tuple individually. However, many practical, real-world problems require a collection of result tuples to satisfy constraints collectively, rather than individually. In this paper, we present *package queries*, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets. We develop a full-fledged package query system, implemented on top of a traditional database engine. Our work makes several contributions. First, we design PaQL, a SQL-based query language that supports the declarative specification of package queries. We prove that PaQL is at least as expressive as integer linear programming, and therefore, evaluation of package queries is in general NP-hard. Second, we present a fundamental evaluation strategy that combines the capabilities of databases and constraint optimization solvers to derive solutions to package queries. The core of our approach is a set of translation rules that transform a package query to an integer linear program. Third, we introduce an offline data partitioning strategy allowing query evaluation to scale to large data sizes. Fourth, we introduce SKETCHREFINE, a scalable algorithm for package evaluation, with strong approximation guarantees ($(1\pm\varepsilon)^6$-factor approximation). Finally, we present extensive experiments over real-world and benchmark data. The results demonstrate that SKETCHREFINE is effective at deriving high-quality package results, and achieves runtime performance that is an order of magnitude faster than directly using ILP solvers over large datasets.

1. INTRODUCTION

Traditional database queries follow a simple model: they define constraints, in the form of selection predicates, that each tuple in the result must satisfy. This model is computationally efficient, as the database system can evaluate each tuple individually to determine whether it satisfies the query conditions. However, many practical, real-world problems require a collection of result tuples to satisfy constraints collectively, rather than individually.

Proceedings of the VLDB Endowment, Vol. 9, No. 7

EXAMPLE 1 (MEAL PLANNER). *A dietitian needs to design a daily meal plan for a patient. She wants a set of three gluten-free meals, between 2,000 and 2,500 calories in total, and with a low total intake of saturated fats.*

EXAMPLE 2 (NIGHT SKY). *An astrophysicist is looking for rectangular regions of the night sky that may potentially contain previously unseen quasars. Regions are explored if their overall redshift is within some specified parameters, and ranked according to their likelihood of containing a quasar [17].*

In these examples, there are some conditions that can be verified on individual data items (e.g., gluten content in a meal), while others need to be evaluated on a collection of items (e.g., total calories). Similar scenarios arise in a variety of application domains, such as investment planning, product bundles, course selection [25], team formation [2, 21], vacation and travel planning [8], and computational creativity [27]. Despite the clear application need, database systems do not currently offer support for these problems, and existing work has focused on application- and domain-specific approaches [2, 8, 21, 25].

In this paper, we present a domain-independent, database-centric approach to address these challenges: We introduce a full-fledged system that supports *package queries*, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets. Package queries are defined over traditional relations, but return *packages*. A package is a collection of tuples that (a) individually satisfy *base predicates* (traditional selection predicates), and (b) collectively satisfy *global predicates* (package-specific predicates). Package queries are combinatorial in nature: the result of a package query is a (potentially infinite) set of packages, and an *objective criterion* can define a preference ranking among them.

Extending traditional database functionality to provide support for packages, rather than supporting packages at the application level, is justified by two reasons: First, the features of packages and the algorithms for constructing them are not unique to each application; therefore, the burden of package support should be lifted off application developers, and database systems should support package queries like traditional queries. Second, the data used to construct packages typically resides in a database system, and packages themselves are structured data objects that should naturally be stored in and manipulated by a database system.

Our work in this paper addresses three important challenges:

1. **Declarative specification of packages.** SQL enables the declarative specification of properties that result tuples should satisfy. In Example 1, it is easy to specify the exclusion of meals with gluten using a regular selection predicate in SQL. However, it is difficult to specify global constraints (e.g., total calories of a set of meals

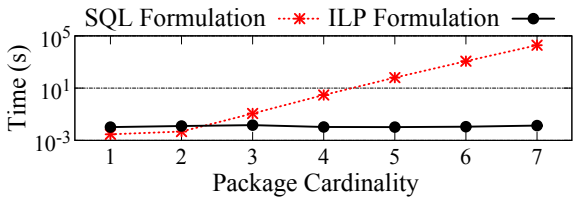

Figure 1: Traditional database technology is ineffective at package evaluation, and the runtime of the naïve SQL formulation (Section 2) of a package query grows exponentially. In contrast, tools such as ILP solvers are more effective.

should be between 2,000 and 2,500 calories). Expressing such a query in SQL requires either complex self-joins that explode the size of the query, or recursion, which results in extremely complex queries that are hard to specify and optimize (Section 2). Our goal is to maintain the declarative power of SQL, while extending its expressiveness to allow for the easy specification of packages.

2. **Evaluation of package queries.** Due to their combinatorial complexity, package queries are harder to evaluate than traditional database queries [9]. Package queries are in fact as hard as integer linear programs (ILP) (Section 2.2). Existing database technology is ineffective at evaluating package queries, even if one were to express them in SQL. Figure 1 shows the performance of evaluating a package query expressed as a multi-way self-join query in traditional SQL (described in detail in Section 2). As the cardinality of the package increases, so does the number of joins, and the runtime quickly becomes prohibitive: In a small set of 100 tuples from the Sloan Digital Sky Survey dataset [28], SQL evaluation takes almost 24 hours to construct a package of 7 tuples. Our goal is to extend the database evaluation engine to take advantage of external tools, such as ILP solvers, which are more effective for combinatorial problems.

3. **Performance and scaling to large datasets.** Integer programming solvers have two major limitations: they require the entire problem to fit in main memory, and they fail when the problem is too complex (e.g., too many variables and/or too many constraints). Our goal is to overcome these limitations through sophisticated evaluation methods that allow solvers to scale to large data sizes.

In this paper, we address these challenges by designing language and algorithmic support for package query specification and evaluation. Specifically, we make the following contributions.

- We present PaQL (Package Query Language), a declarative language that provides simple extensions to standard SQL to support constraints at the package level. We prove that PaQL is at least as expressive as integer linear programming, which implies that evaluation of package queries is NP-hard (Section 2).
- We present a fundamental evaluation strategy DIRECT that combines the capabilities of databases and constraint optimization solvers to derive solutions to package queries. The core of our approach is a set of translation rules that transform a package query to an integer linear program. This translation allows for the use of highly-optimized external tools for the evaluation of package queries (Section 3).
- We introduce an offline data partitioning strategy that allows package query evaluation to scale to large data sizes. The core of our evaluation strategy SKETCHREFINE lies on separating the package computation into multiple stages, each with small subproblems, which the solver can evaluate efficiently. In the first stage, the algorithm "sketches" an initial sample package from a set of representative tuples, while the subsequent stages "refine" the current package by solving an ILP within each partition. SKETCHREFINE guarantees a $(1 \pm \varepsilon)^6$-factor approximation for the package results compared to DIRECT (Section 4).
- We present an extensive experimental evaluation on both real-world data and the TPC-H benchmark (Section 5) that shows that our query evaluation method SKETCHREFINE: (1) is able to produce packages an order of magnitude faster than the ILP solver used directly on the entire problem; (2) scales up to sizes that the solver cannot manage directly; (3) produces packages of very good quality in terms of objective value; (4) is robust to partitioning built in anticipation of different workloads.

2. LANGUAGE SUPPORT FOR PACKAGES

Data management systems do not natively support package queries. While there are ways to express package queries in SQL, these are cumbersome and inefficient.

Specifying packages with self-joins. When packages have strict cardinality (number of tuples), and only in this case, it is possible to express package queries using traditional self-joins. For instance, self-joins can express the query of Example 1 as follows:

```
SELECT *   FROM Recipes R1, Recipes R2, Recipes R3
WHERE      R1.pk < R2.pk AND R2.pk < R3.pk AND
  R1.gluten = 'free' AND R2.gluten = 'free' AND R3.gluten = 'free'
  AND R1.kcal + R2.kcal + R3.kcal BETWEEN 2.0 AND 2.5
ORDER BY  R1.saturated_fat + R2.saturated_fat + R3.saturated_fat
```

This query is efficient only for constructing packages with very small cardinality: larger cardinality requires a larger number of self-joins, quickly rendering evaluation time prohibitive (Figure 1). The benefit of this specification is that the optimizer can use the traditional relational algebra operators, and augment its decisions with package-specific strategies. However, this method does not apply for packages of unbounded cardinality.

Using recursion in SQL. More generally, SQL can express package queries by generating and testing each possible subset of the input relation. This requires recursion to build a *powerset table*; checking each set in the powerset table for the query conditions will yield the result packages. This approach has three major drawbacks. First, it is not declarative, and the specification is tedious and complex. Second, it is not amenable to optimization in existing systems. Third, it is extremely inefficient to evaluate, because the powerset table generates an exponential number of candidates.

2.1 PaQL: The Package Query Language

Our goal is to support package specification in a declarative and intuitive way. In this section, we describe PaQL, a declarative query language that introduces simple extensions to SQL to define package semantics and package-level constraints. We first show how PaQL can express the query of Example 1, as our running example, to demonstrate the new language features:

```
Q: SELECT       PACKAGE(R) AS P
   FROM         Recipes R REPEAT 0
   WHERE        R.gluten = 'free'
   SUCH THAT    COUNT(P.*) = 3 AND
                SUM(P.kcal) BETWEEN 2.0 AND 2.5
   MINIMIZE     SUM(P.saturated_fat)
```

Basic semantics. The new keyword PACKAGE differentiates PaQL from traditional SQL queries.

```
Q1: SELECT  *              Q2: SELECT  PACKAGE(R) AS P
    FROM    Recipes R          FROM    Recipes R
```

The semantics of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are fundamentally different: $\mathcal{Q}_1$ is a traditional SQL query, with a unique, finite result set (the entire Recipes table), whereas there are infinitely many packages that satisfy the package query $\mathcal{Q}_2$: all possible multisets of tuples from the input relation. The result of a package query like $\mathcal{Q}_2$ is a set of packages. Each package resembles a relational table containing a collection of tuples (with possible repetitions) from relation Recipes, and therefore a package result of $\mathcal{Q}_2$ follows the schema of Recipes. PaQL syntax permits multiple relations in the FROM clause; in that case, the packages produced will follow the schema of the join result.

In the remainder of this paper, we focus on package queries without joins. This is for two reasons: (1) The join operation is part of traditional SQL and can occur before package-specific computations. (2) There are important implications in the consideration of joins that extend beyond the scope of our work. Specifically, materializing the join result is not always necessary, but rather, there are space-time tradeoffs and system-level solutions that can improve query performance in the presence of joins. Section 4.5 offers a high-level discussion of these points, but these extensions are orthogonal to the techniques we present in this work.

Although semantically valid, a query like $\mathcal{Q}_2$ would not occur in practice, as most application scenarios expect few, or even exactly one result. We proceed to describe the additional constraints in the example query $\mathcal{Q}$ that restrict the number of package results.

Repetition constraint. The REPEAT 0 statement in query $\mathcal{Q}$ specifies that no tuple from the input relation can appear multiple times in a package result. If this restriction is absent (as in query $\mathcal{Q}_2$), tuples can be repeated an unlimited number of times. By allowing no repetitions, $\mathcal{Q}$ restricts the package space from infinite to 2^n, where n is the size of the input relation. Generalizing, the specification REPEAT $\mathcal{K}$ allows a package to repeat tuples up to $\mathcal{K}$ times, resulting in $(2+\mathcal{K})^n$ candidate packages.

Base and global predicates. A package query defines two types of predicates. A *base predicate*, defined in the WHERE clause, is equivalent to a selection predicate and can be evaluated with standard SQL: any tuple in the package needs to *individually* satisfy the base predicate. For example, query $\mathcal{Q}$ specifies the base predicate: R.gluten = 'free'. Since base predicates directly filter input tuples, they are specified over the input relation R. *Global predicates* are the core of package queries, and they appear in the new SUCH THAT clause. Global predicates are higher-order than base predicates: they cannot be evaluated on individual tuples, but on tuple collections. Since they describe package-level constraints, they are specified over the package result P, e.g., COUNT(P.*) = 3, which limits the query results to packages of exactly 3 tuples.

The global predicates shown in query $\mathcal{Q}$ abbreviate aggregates that are in reality subqueries. For example, COUNT(P.*) = 3, is an abbreviation for (SELECT COUNT(*) FROM P) = 3. Using subqueries, PaQL can express arbitrarily complex global constraints among aggregates over a package.

Objective clause. The objective clause specifies a ranking among candidate package results, and appears with either the MINIMIZE or MAXIMIZE keyword. It is a condition on the package-level, and hence it is specified over the package result P, e.g., MINIMIZE SUM(P.saturated_fat). Similarly to global predicates, this form is a shorthand for MINIMIZE (SELECT SUM(saturated_fat) FROM P). A PaQL query with an objective clause returns a single result: the package that optimizes the value of the objective. The evaluation methods that we present in this work focus on such queries. In prior work [6], we described preliminary techniques for returning multiple packages in the absence of optimization objectives, but a thorough study of such methods is left to future work.

While PaQL can use arbitrary aggregate functions in the global predicates and the objective clause, in this work, we assume that package queries are limited to *linear* functions. We defer the study of non-linear functions and UDFs to future work.

2.2 Expressiveness and complexity of PaQL

Package queries are at least as hard as integer linear programs, as the following theoretical results establish.

THEOREM 1 (EXPRESSIVENESS OF PAQL). *Every integer linear program can be expressed as a package query in PaQL.*

We include the proofs of our theoretical results in the full version of the paper [5]. At a high level, the proof employs a reduction from an integer linear program to a PaQL query. The reduction maps the linear constraints and objective into the corresponding PaQL clauses, using the constraint coefficients to generate the input relation for the package query. As a direct consequence of Theorem 1, we also obtain the following result about the complexity of evaluating package queries.

COROLLARY 2 (COMPLEXITY OF PACKAGE QUERIES). *Package queries are NP-hard.*

In Section 3, we extend the result of Theorem 1 to also show that every PaQL query that does not contain non-linear functions can be expressed as an integer linear program, through a set of translation rules. This transformation is the first step in package evaluation, but, due to the limitations of ILP solvers, it is not efficient or scalable in practice. To make package evaluation practical, we develop SKETCHREFINE, a technique that augments the ILP transformation with a partitioning mechanism, allowing package evaluation to scale to large datasets (Section 4).

3. ILP FORMULATION

In this section, we present an ILP formulation for package queries. This formulation is at the core of our evaluation methods DIRECT and SKETCHREFINE. The results presented in this section are inspired by the translation rules employed by Tiresias [22] to answer *how-to queries*. However, there are several important differences between how-to and package queries, which we discuss extensively in the overview of the related work (Section 6).

3.1 PaQL to ILP Translation

Let R indicate the input relation, $n = |\mathsf{R}|$ the number of tuples in R, R.attr an attribute of R, P a package, f a linear aggregate function (such as COUNT and SUM), $\odot \in \{\leq, \geq\}$ a constraint inequality, and $v \in \mathbb{R}$ a constant. For each tuple t_i from R, $1 \leq i \leq n$, the ILP problem includes a nonnegative integer variable x_i ($x_i \geq 0$), indicating the number of times t_i is included in an answer package. We also use $\bar{x} = \langle x_1, x_2, \ldots, x_n \rangle$ to denote the vector of all integer variables. A PaQL query is formulated as an ILP problem using the following translation rules:

1. **Repetition constraint.** The REPEAT keyword, expressible in the FROM clause, restricts the domain that the variables can take on. Specifically, REPEAT $\mathcal{K}$ implies $0 \leq x_i \leq \mathcal{K}+1$.
2. **Base predicate.** Let β be a base predicate, e.g., R.gluten = 'free', and R_β the relation containing tuples from R satisfying β. We encode β by setting $x_i = 0$ for every tuple $t_i \notin \mathsf{R}_\beta$.
3. **Global predicate.** Each global predicate in the SUCH THAT clause takes the form $f(\mathsf{P}) \odot v$. For each such predicate, we derive a linear function $f'(\bar{x})$ over the integer variables. A cardinality constraint $f(\mathsf{P}) = \mathsf{COUNT}(\mathsf{P}.*)$ is linearly translated into

$f'(\vec{x}) = \sum_i x_i$. A summation constraint $f(\mathsf{P}) = \mathsf{SUM}(\mathsf{P.attr})$ is linearly translated into $f'(\vec{x}) = \sum_i (t_i.\mathsf{attr})x_i$. We further illustrate the translation with two non-trivial examples:

- $\mathsf{AVG}(\mathsf{P.attr}) \leq v$ is translated as

$$\sum_i (t_i.\mathsf{attr})x_i / \sum_i x_i \leq v \;\equiv\; \sum_i (t_i.\mathsf{attr} - v)x_i \leq 0$$

- (SELECT COUNT$(*)$ FROM P WHERE P.carbs > 0) $\geq$ (SELECT COUNT$(*)$ FROM P WHERE P.protein ≤ 5) is translated as

$$\begin{array}{l} \mathsf{R_c} := \{t_i \in \mathsf{R} \mid t_i.\mathsf{carbs} > 0\} \\ \mathsf{R_p} := \{t_i \in \mathsf{R} \mid t_i.\mathsf{protein} \leq 5\} \\ \mathbb{1}_{\mathsf{R_c}}(t_i) := 1 \text{ if } t_i \in \mathsf{R_c};\ 0 \text{ otherwise.} \\ \mathbb{1}_{\mathsf{R_p}}(t_i) := 1 \text{ if } t_i \in \mathsf{R_p};\ 0 \text{ otherwise.} \\ \sum_i (\mathbb{1}_{\mathsf{R_c}}(t_i) - \mathbb{1}_{\mathsf{R_p}}(t_i))x_i \geq 0 \end{array}$$

General Boolean expressions over the global predicates can be encoded into a linear program with the help of Boolean variables and linear transformation tricks found in the literature [4].

4. **Objective clause.** We encode MAXIMIZE $f(\mathsf{P})$ as $\max f'(\vec{x})$, where $f'(\vec{x})$ is the encoding of $f(\mathsf{P})$. Similarly MINIMIZE $f(\mathsf{P})$ is encoded as $\min f'(\vec{x})$. If the query does not include an objective clause, we add the *vacuous* objective $\max \sum_i 0 \cdot x_i$.

We call the relations $\mathsf{R_\beta}$, $\mathsf{R_c}$, and $\mathsf{R_p}$ described above *base relations*. This formulation, together with Theorem 1, shows that single-relation package queries with linear constraints correspond exactly to ILP problems.

3.2 Query Evaluation with DIRECT

Using the ILP formulation, we develop our basic evaluation method for package queries, called DIRECT. We later extend this technique to our main algorithm, SKETCHREFINE, which supports efficient package evaluation in large data sets (Section 4).

Package evaluation with DIRECT employs three simple steps:

1. **ILP formulation.** We transforms a PaQL query to an ILP problem using the rules described in Section 3.1.
2. **Base relations.** We compute the base relations, such as $\mathsf{R_\beta}$, $\mathsf{R_c}$, and $\mathsf{R_p}$, with a series of standard SQL queries, one for each, or by simply scanning R once and populating these relations simultaneously. After this phase, all variables x_i such that $x_i = 0$ can be eliminated from the ILP problem because the corresponding tuple t_i cannot appear in any package solution. This can significantly reduce the size of the problem.
3. **ILP execution.** We employ an off-the-shelf ILP solver as a black box to get a solution $\vec{x}^*$ for all the integer variables x_i of the problem. Each x_i^* informs the number of times tuple t_i should be included in the answer package.

The DIRECT algorithm has two crucial drawbacks. First, it is only applicable if the input relation is small enough to fit entirely in main memory: ILP solvers, such as IBM's CPLEX, require the entire problem to be loaded in memory before execution. Second, even for problems that fit in main memory, this approach may fail due to the complexity of the integer problem. In fact, integer linear programming is a notoriously hard problem, and modern ILP solvers use algorithms, such as *branch-and-cut* [24], that often perform well in practice, but can "choke" even on small problem sizes due to their exponential worst-case complexity [7]. This may result in unreasonable performance due to solvers using too many resources (main memory, virtual memory, CPU time), eventually thrashing the entire system.

Algorithm 1 Scalable Package Query Evaluation

```
1: procedure SketchRefine(𝒫:Partitioning, 𝒬:Package Query)
2:     p_S ← Sketch(𝒫, 𝒬)
3:     if failure then
4:         return infeasible
5:     else
6:         p ← Refine(p_S, 𝒫, 𝒬)
7:         if failure then
8:             return infeasible
9:         else
10:            return p
```

4. SCALABLE PACKAGE EVALUATION

In this section, we present SKETCHREFINE, an approximate divide-and-conquer evaluation technique for efficiently answering package queries on large datasets. SKETCHREFINE smartly decomposes a query into smaller queries, formulates them as ILP problems, and employs an ILP solver as a "black box" evaluation method to answer each individual query. By breaking down the problem into smaller subproblems, the algorithm avoids the drawbacks of the DIRECT approach. Further, we prove that SKETCHREFINE is guaranteed to always produce feasible packages with an approximate objective value (Section 4.3).

The algorithm is based on an important observation: *similar tuples are likely to be interchangeable within packages*. A group of similar tuples can therefore be "compressed" to a single *representative tuple* for the entire group. SKETCHREFINE *sketches* an initial answer package using only the set of representative tuples, which is substantially smaller than the original dataset. This initial solution is then *refined* by evaluating a subproblem for each group, iteratively replacing the representative tuples in the current package solution with original tuples from the dataset. Figure 2 provides a high-level illustration of the three main steps of SKETCHREFINE:

1. **Offline partitioning (Section 4.1).**The algorithm assumes a partitioning of the data into groups of similar tuples. This partitioning is performed offline (not at query time), and our experiments show that SKETCHREFINE remains very effective even with partitionings that do not match the query workload (Section 5.2.3). In our implementation, we partition data using k-dimensional quad trees [11], but other partitioning schemes are possible.
2. **Sketch (Section 4.2.1).** SKETCHREFINE sketches an initial package by evaluating the package query only over the set of representative tuples.
3. **Refine (Section 4.2.2).** Finally, SKETCHREFINE transforms the initial package into a complete package by replacing each representative tuple with some of the original tuples from the same group, one group at a time.

SKETCHREFINE always constructs *approximate feasible* packages, i.e., packages that satisfy all the query constraints, but with a possibly sub-optimal objective value that is guaranteed to be within certain approximation bounds (Section 4.3). SKETCHREFINE may suffer from *false infeasibility*, which happens when the algorithm reports a feasible query to be infeasible. The probability of false infeasibility is, however, low and bounded (Section 4.4).

In the subsequent discussion, we use R to denote the input relation of n tuples, $t_i \in \mathsf{R}$, $1 \leq i \leq n$. R is partitioned into m groups $G_1, \ldots, G_m$. Each group G_j, $1 \leq j \leq m$, has a representative tuple $\tilde{t}_j$, which may not always appear in R. We denote the partitioned space with $\mathcal{P} = \{(G_j, \tilde{t}_j) \mid 1 \leq j \leq m\}$. We refer to packages that contain some representative tuples as *sketch packages* and packages with only original tuples as *complete packages* (or simply *packages*).

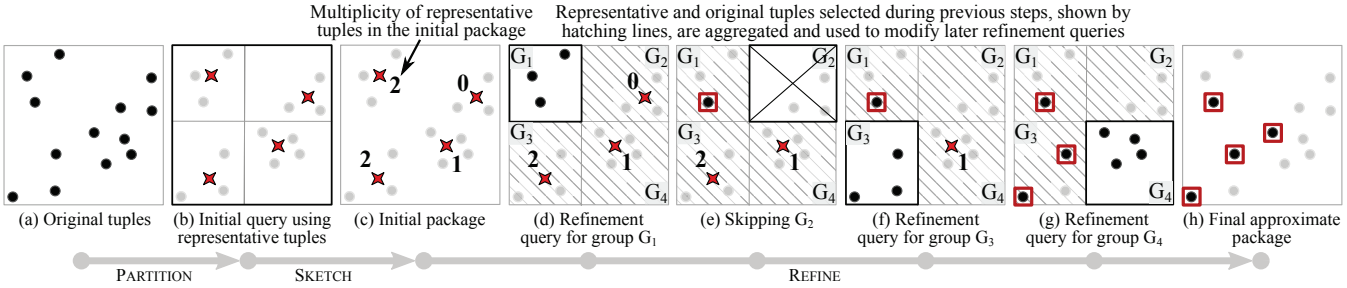

Figure 2: The original tuples (a) are partitioned into four groups and a representative is constructed for each group (b). The initial sketch package (c) contains only representative tuples, with possible repetitions up the size of each group. The refine query for group G_1 (d) involves the original tuples from G_1 and the aggregated solutions to all other groups (G_2, G_3, and G_4). Group G_2 can be skipped (e) because no representatives could be picked from it. Any solution to previously refined groups are used while refining the solution for the remaining groups (f and g). The final approximate package (h) contains only original tuples.

We denote a complete package with p and a sketch package with $p_{\mathcal{S}}$, where $\mathcal{S} \subseteq \mathcal{P}$ is the set of groups that are yet to be refined to transform $p_{\mathcal{S}}$ into a complete answer package p.

4.1 Offline Partitioning

SKETCHREFINE relies on an offline partitioning of the input relation R into groups of similar tuples. Partitioning is based on a set of k numerical *partitioning attributes*, $\mathcal{A}$, from the input relation R, and uses two parameters: a *size* threshold and a *radius* limit.

DEFINITION 1 (SIZE THRESHOLD, τ). *The* size threshold τ, $1 \leq \tau \leq n$, *restricts the size of each partitioning group* G_j, $1 \leq j \leq m$, *to a maximum of* τ *original tuples, i.e.,* $|G_j| \leq \tau$.

DEFINITION 2 (RADIUS LIMIT, ω). *The* radius $r_j \geq 0$ *of a group* G_j *is the greatest absolute distance between the representative tuple of* G_j, $\tilde{t}_j$, *and every original tuple of the group, across all partitioning attributes:*

$$r_j = \max_{\substack{t_j \in G_j \\ \mathsf{attr} \in \mathcal{A}}} |\tilde{t}_j.\mathsf{attr} - t_j.\mathsf{attr}|$$

The radius limit ω, $\omega \geq 0$, *requires that for every partitioning group* G_j, $1 \leq j \leq m$, $r_j \leq \omega$.

The size threshold, τ, affects the number of clusters, m, as smaller clusters (lower τ) implies more of them (larger m), especially on skewed datasets. As we discuss later (Section 4.2), for best response time of SKETCHREFINE, τ should be set so that both m and τ are small. Our experiments show that a proper setting can yield to an order of magnitude improvement in query response time (Section 5.2.2). The radius limit, ω, should be set according to a desired approximation guarantee (Section 4.3). Note that the same partitioning can be used to support a multitude of queries over the same dataset. In our experiments, we show that a single partitioning performs consistently well across different queries.

Partitioning method. Different methods can be used for partitioning. Our implementation is based on k-dimensional *quad-tree indexing* [11]. The method recursively partitions a relation into groups until all the groups satisfy the size threshold and meet the radius limit. First, relation R is augmented with an extra group ID column gid, such that $t_i.\mathsf{gid} = j$ iff tuple t_i is assigned to group G_j. The procedure initially creates a single group G_1 that includes all the original tuples from relation R, by initializing $t_i.\mathsf{gid} = 1$ for all tuples. Then, it recursively proceeds as follows:

- The procedure computes the sizes and radii of the current groups via a query that groups tuples by their gid value. The same group-by query also computes the *centroid* tuple of each group. The centroid is computed by averaging the tuples in the group on each of the partitioning attributes $\mathcal{A}$.
- If group G_j has more tuples than the size threshold, or a radius larger than the radius limit, the tuples in group G_j are partitioned into 2^k subgroups ($k = |\mathcal{A}|$). The group's centroid is used as the pivot point to generate sub-quadrants: tuples that reside in the same sub-quadrant are grouped together.

Our method recursively executes two SQL queries on each subgroup that violates the size or the radius condition. In the last iteration, the last group-by query computes the centroids for each group. These are the representative tuples, $\tilde{t}_j$, $1 \leq j \leq m$, and are stored in a new *representative relation* $\tilde{\mathsf{R}}(\mathsf{gid}, \mathsf{attr}_1, \ldots, \mathsf{attr}_k)$.

Alternative partitioning approaches. We experimented with different clustering algorithms, such as *k-means* [15], *hierarchical clustering* [20] and DBSCAN [10], using off-the-shelf libraries such as *Scikit-learn* [26]. Existing clustering algorithms present various problems: First, they tend to vary substantially in the properties of the generated clusters. In particular, none of the existing clustering techniques can natively generate clusters that satisfy the size threshold τ and radius limit ω. In fact, most of the clustering algorithms take as input the *number of clusters* to generate, without offering any means to restrict the size of each cluster nor their radius. Second, existing implementations only support in-memory cluster computation, and DBMS-oriented implementations usually need complex and inefficient queries. On the other hand, space partitioning techniques from multi-dimensional indexing, such as *k-d trees* [3] and *quad trees* [11], can be more easily adapted to satisfy the size and radius conditions, and to work within the database: our partitioning method works directly on the input table via simple SQL queries.

One-time cost. Partitioning is an expensive procedure. To avoid paying its cost at query time, the dataset is partitioned in advance and used to answer a workload of package queries. For a known workload, our experiments show that partitioning the dataset on the union of all query attributes provides the best performance in terms of query evaluation time and approximation error for the computed answer package (Section 5.2.3). We also demonstrate that our query evaluation approach is robust to a wide range of partition sizes, and to imperfect partitions that cover more or fewer attributes than those used in a particular query. This means that, even without a known workload, a partitioning performed on all of the data attributes still provides good performance.

The radius limit is necessary for the theoretical guarantee of the approximation bounds (Section 4.3). However, we show empirically that partitioning satisfying the size threshold alone produces satisfac-

tory answers while reducing the offline partitioning cost: Meeting a size threshold requires fewer partitioning iterations than meeting a radius limit especially if the dataset is sparse across the attribute domains (Section 5).

Dynamic partitioning. In our implementation of SKETCHREFINE, the choice of partitioning is static. Our technique also works with a dynamic approach to partitioning: by maintaining the entire hierarchical structure of the quad-tree index, one can traverse the index at query time to generate the coarsest partitioning that satisfies the required radius condition. However, our empirical results show that this approach incurs unnecessary overhead, as static partitioning already performs extremely well in practice (Section 5).

4.2 Query Evaluation with SKETCHREFINE

During query evaluation, SKETCHREFINE first *sketches* a package solution using the representative tuples (SKETCH), and then it *refines* it by replacing representative tuples with original tuples (REFINE). We describe these steps using the example query $\mathcal{Q}$ from Section 2.1.

4.2.1 SKETCH

Using the representative relation $\tilde{\mathsf{R}}$ (Section 4.1), the SKETCH procedure constructs and evaluates a *sketch query*, $\mathcal{Q}[\tilde{\mathsf{R}}]$. The result is an initial sketch package, $p_{\mathcal{S}}$, containing representative tuples that satisfy the same constraints as the original query $\mathcal{Q}$:

$\mathcal{Q}[\tilde{\mathsf{R}}]$: SELECT PACKAGE($\tilde{\mathsf{R}}$) AS $p_{\mathcal{S}}$
FROM $\tilde{\mathsf{R}}$
WHERE $\tilde{\mathsf{R}}$.gluten = 'free'
SUCH THAT
COUNT($p_{\mathcal{S}}$.*) = 3 AND
SUM($p_{\mathcal{S}}$.kcal) BETWEEN 2.0 AND 2.5 AND
(**SELECT COUNT**(*) **FROM** $\mathbf{p_{\mathcal{S}}}$ **WHERE gid** $= \mathbf{1}$) $\leq |\mathbf{G_1}|$
AND ...
(**SELECT COUNT**(*) **FROM** $\mathbf{p_{\mathcal{S}}}$ **WHERE gid** $= \mathbf{m}$) $\leq |\mathbf{G_m}|$
MINIMIZE SUM($p_{\mathcal{S}}$.saturated_fat)

The new global constraints, highlighted in bold, ensure that every representative tuple does not appear in $p_{\mathcal{S}}$ more times than the size of its group, G_j. This accounts for the repetition constraint REPEAT 0 in the original query. Generalizing, with REPEAT $\mathcal{K}$, each $\tilde{t}_j$ can be repeated up to $|G_j|(1+\mathcal{K})$ times. These constraints are simply omitted from $\mathcal{Q}[\tilde{\mathsf{R}}]$ if the original query does not contain a repetition constraint.

Since the representative relation $\tilde{\mathsf{R}}$ contains exactly m representative tuples, the ILP problem corresponding to this query has only m variables. This is typically small enough for the black box ILP solver to manage directly, and thus we can solve this package query using the DIRECT method (Section 3.2). If m is too large, we can solve this query *recursively* with SKETCHREFINE: the set of m representatives is further partitioned into smaller groups until the subproblems reach a size that can be efficiently solved directly.

The SKETCH procedure *fails* if the sketch query $\mathcal{Q}[\tilde{\mathsf{R}}]$ is infeasible, in which case SKETCHREFINE reports the original query $\mathcal{Q}$ as infeasible (Algorithm 1). This may constitute *false infeasibility*, if $\mathcal{Q}$ is actually feasible. In Section 4.4, we show that the probability of false infeasibility is low and bounded, and we present simple methods to avoid this outcome.

4.2.2 REFINE

Using the sketched solution over the representative tuples, the REFINE procedure iteratively replaces the representative tuples with tuples from the original relation R, until no more representatives are present in the package. The algorithm *refines* the sketch package $p_{\mathcal{S}}$, one group at a time: For a group G_j with representative $\tilde{t}_j \in p_{\mathcal{S}}$, the algorithm derives package $\bar{p}_j$ from $p_{\mathcal{S}}$ by eliminating all instances of $\tilde{t}_j$; it then seeks to replace the eliminated representatives with actual tuples, by issuing a *refine query*, $\mathcal{Q}[G_j]$, on group G_j:

$\mathcal{Q}[G_j]$: SELECT PACKAGE(G_j) AS p_j
FROM G_j REPEAT 0
WHERE G_j.gluten = 'free'
SUCH THAT
COUNT(p_j.*) + **COUNT**($\bar{p}_j$.*) = 3 AND
SUM(p_j.kcal) + **SUM**($\bar{p}_j$.kcal) BETWEEN 2.0 AND 2.5
MINIMIZE SUM(p_j.saturated_fat)

The query derives a set of tuples p_j, as a replacement for the occurrences of the representatives of G_j in $p_{\mathcal{S}}$. The global constraints in $\mathcal{Q}[G_j]$ ensure that the combination of tuples in p_j and $\bar{p}_j$ satisfy the original query $\mathcal{Q}$. Thus, this step produces the new *refined sketch package* $p'_{\mathcal{S}'} = \bar{p}_j \cup p_j$, where $\mathcal{S}' = \mathcal{S} \setminus \{(G_j, \tilde{t}_j)\}$.

Since G_j has at most τ tuples, the ILP problem corresponding to $\mathcal{Q}[G_j]$ has at most τ variables. This is typically small enough for the black box ILP solver to solve directly, and thus we can solve this package query using the DIRECT method (Section 3.2). Similarly to the sketch query, if τ is too large, we can solve this query recursively with SKETCHREFINE: the tuples in group G_j are further partitioned into smaller groups until the subproblems reach a size that can be efficiently solved directly.

Ideally, the REFINE step will only process each group with representatives in the initial sketch package once. However, the order of refinement matters as each refinement step is greedy: it selects tuples to replace the representatives of a single group, without considering the effects of this choice on other groups. As a result, a particular refinement step may render the query infeasible (no tuples from the remaining groups can satisfy the constraints). When this occurs, REFINE employs a *greedy backtracking* strategy that reconsiders groups in a different order.

Greedy backtracking. REFINE activates backtracking when it encounters an infeasible *refine query*, $\mathcal{Q}[G_j]$. Backtracking *greedily prioritizes* the infeasible groups. This choice is motivated by a simple heuristic: if the refinement on G_j fails, it is likely due to choices made by previous refinements; therefore, by prioritizing G_j, we reduce the impact of other groups on the feasibility of $\mathcal{Q}[G_j]$. This heuristic does not affect the approximation guarantees (Section 4.3).

Algorithm 2 details the REFINE procedure. The algorithm logically traverses a *search tree* (which is only constructed as new branches are created and new nodes visited), where each node corresponds to a unique sketch package $p_{\mathcal{S}}$. The traversal starts from the *root*, corresponding to the initial sketch package, where no groups have been refined ($\mathcal{S} = \mathcal{P}$), and finishes at the first encountered *leaf*, corresponding to a complete package ($\mathcal{S} = \emptyset$). The algorithm terminates as soon as it encounters a complete package, which it returns (line 3). The algorithm assumes a (initially random) refinement order for all groups in $\mathcal{S}$, and places them in a priority queue (line 6). During refinement, this group order can change by prioritizing groups with infeasible refinements (line 24).

Run time complexity. In the best case, all refine queries are feasible and the algorithm never backtracks. In this case, the algorithm makes up to m calls to the ILP solver to solve problems of size up to τ, one for each refining group. In the worst case, SKETCHREFINE tries every group ordering leading to an exponential number of calls to the ILP solver. Our experiments show that the best case is the most common and backtracking occurs infrequently.

4.3 Approximation Guarantees

SKETCHREFINE provides strong theoretical guarantees. We prove that for a desired approximation parameter ε, we can derive a radius limit ω for the offline partitioning that guarantees

Algorithm 2 Greedy Backtracking Refinement

```
1: procedure REFINE(p_S, P, Q)
2:     if S = ∅ then                          ▷ Base case: all groups already refined
3:         return p_S
4:     F ← ∅                                                          ▷ Failed groups
5:     ▷ Arrange S in some initial order (e.g., random)
6:     U ← priorityQueue(S)
7:     while U ≠ ∅ do
8:         (G_j, t̃_j) ← dequeue(U)
9:         ▷ Skip groups that have no representative in p_S
10:        if t̃_j ∉ p_S then
11:            continue
12:        p'_S' ← Refine p_S on group G_j
13:        if Q[G_j] is infeasible then
14:            if S ≠ P then                  ▷ If p_S is not the initial package
15:                ▷ Greedily backtrack with non-refinable group
16:                F ← F ∪ {(G_j, t̃_j)}
17:                return failure(F)
18:        else
19:            ▷ Greedily recurse with refinable group
20:            p ← REFINE(p'_S', P, Q)
21:            if failure(F') then
22:                F ← F ∪ F'
23:                ▷ Greedily prioritize non-refinable groups
24:                prioritize(U, F)
25:            else
26:                return p
27:    ▷ None of the groups in S can be refined (invariant: F = S)
28:    return failure(F)
```

that SKETCHREFINE will produce a package with objective value $(1 \pm \varepsilon)^6$-factor close to the objective value of the solution generated by DIRECT for the same query.

THEOREM 3 (APPROXIMATION BOUNDS). *For any feasible package query with a maximization (minimization, resp.) objective and approximation parameter ε, $0 \leq \varepsilon < 1$ ($\varepsilon \geq 0$, resp.), any database instance, any set of partitioning attributes $\mathcal{A}$, superset of the numerical query attributes, any size threshold τ, and radius limit:*

$$\omega = \min_{\substack{1 \leq j \leq m \\ \text{attr} \in \mathcal{A}}} \gamma \, |\tilde{t}_j.\text{attr}|, \;\; \textit{where } \gamma = \varepsilon \;\; (\gamma = \tfrac{\varepsilon}{1+\varepsilon}, \textit{ resp.}) \qquad (1)$$

The package produced by SKETCHREFINE *(if any) is guaranteed to have objective value $\geq (1-\varepsilon)^6 OPT$ ($\leq (1+\varepsilon)^6 OPT$, resp.), where OPT is the objective value of the* DIRECT *solution.*

At a high level, the proof includes two steps: In the first step, we prove that the initial sketch package is a $(1 \pm \varepsilon)^3$-approximation with respect to DIRECT; In the second step, we prove that the final package produced by SKETCHREFINE is a $(1 \pm \varepsilon)^3$-approximation with respect to the initial sketch package.

In general, enforcing this radius limit, especially with lower ε values, may cause the resulting partitions to become excessively small. While still obeying the approximation guarantees, this could increase the number of resulting partitions and thus degrade the running time performance of SKETCHREFINE. In the worst case, SKETCHREFINE could degenerate into DIRECT. This is an important trade-off between running time and quality that we have observed in our experimental data as well, and it is a very common characteristic of most approximation schemes [30].

4.4 The Case of False Infeasibility

For a feasible query Q, false negatives, or *false infeasibility*, may happen in two cases: (1) when the sketch query $Q[\tilde{R}]$ is infeasible; (2) when greedy backtracking fails (possibly due to suboptimal partitioning). In both cases, SKETCHREFINE would (incorrectly) report a feasible package query as infeasible. False negatives are, however, extremely rare, as the following theorem establishes.

THEOREM 4 (FALSE INFEASIBILITY). *For any feasible package query, any database instance, any set of partitioning attributes $\mathcal{A}$ that is a superset of the query attributes, any size threshold τ, and any radius limit ω,* SKETCHREFINE *finds a feasible package with high probability that inversely depends on query selectivity.*

The *selectivity* of a package query denotes the probability of a random package being infeasible (thus, lower selectivity implies higher probability of a random package being feasible). To prove this result, we show that if a random package is feasible, then with high probability the sketch query and all refine queries are feasible. Thus, lower selectivity implies higher probability that SKETCHREFINE successfully terminates with a feasible package.

We discuss four potential ways to deal with false infeasibility, which we plan to explore in future work:

1. **Hybrid sketch query.** A simple method to avoid false infeasibility in the SKETCH step is to merge the sketch query $Q[\tilde{R}]$ with one of the refine queries. This "hybrid" sketch query would select original tuples for one of the groups, and at the same time select representative tuples for the remaining groups. Groups can be tried in any order (e.g., in random order), until one of the hybrid sketch queries is feasible.

2. **Further partitioning.** In some cases, the centroid of a group may not be a very good representative (e.g., when the data is skewed). This can sometimes result in false negatives. Further partitioning by reducing the size threshold τ may eliminate the problem.

3. **Dropping partitioning attributes.** A more principled approach consists of "projecting" the partitioning onto fewer dimensions by reducing the number of partitioning attributes. In doing so, some groups merge, increasing the chance that previously infeasible groups become feasible. The choice of attributes to remove from the partitioning can be guided by the last infeasible ILP problem. Most ILP solvers provide functionality to identify a *minimal set of infeasible constraints*: removing any constraint from the set makes the problem feasible.[1] Removing the attributes that participate in these constraints from the partitioning can increase the odds of discovering a feasible solution.

4. **Iterative group merging.** In a brute-force approach, we can merge groups iteratively, until the sub-queries become feasible. In the worst case, this process reduces the problem to the original problem (i.e., with no partitioning), and thus it is guaranteed to find a solution to any feasible query, at the cost of performance.

4.5 Discussion

SKETCHREFINE is an evaluation strategy for package queries with three important advantages. First, it scales naturally to very large datasets, by breaking down the problem into smaller, manageable subproblems, whose solutions can be iteratively combined to form the final result. Second, it provides flexible approximations with strong theoretical guarantees on the quality of the package results. Third, while our current implementation of SKETCHREFINE employs ILP solvers to evaluate the generated subproblems, our algorithm can use any other black box solution for package queries, even solutions that work entirely in main memory, and whose efficiency drastically degrades with larger problem sizes. We plan to explore these alternatives in our future work.

[1] This set is usually referred to as *irreducible infeasible set* (IIS).

Parallelizing SKETCHREFINE. Its data partitioning and problem division strategies give SKETCHREFINE great potential for parallelization. However, the proper parallelization strategy is non-obvious, and is a nontrivial part of future work. A simple parallelization strategy could perform refinement on several groups in parallel. However, since refinements make local decisions, this process is more likely to reach infeasibility, requiring costly backtracking steps and resulting in wasted computation. Alternatively, parallelization may focus on the backtracking process, using additional resources to evaluate different group orderings in parallel.

Handling joins. In this paper, we focused on package queries over single relations. Handling joins poses interesting implications, orthogonal to the evaluation techniques that we presented in this work. In the presence of joins, the system can simply evaluate and materialize the join result before applying the package-specific transformations. However, the materialization of the join result is not always necessary: DIRECT generates variables through a single sequential scan of the join result, and thus the join tuples can be pipelined into the ILP generation without being materialized. However, not materializing the join results means that some of the join tuples will need to be recomputed to populate the solution package. Therefore, there is a space-time tradeoff in the consideration of materializing the join. Further, this tradeoff can be improved with hybrid, system-level solutions, such as storing the record IDs of joining tuples to enable faster access during package generation. These considerations are well-beyond our current scope, and are orthogonal to the techniques that we present in this work.

5. EXPERIMENTAL EVALUATION

In this section, we present an extensive experimental evaluation of our techniques for package query execution, both on real-world and on benchmark data. Our results show the following properties of our methods: (1) SKETCHREFINE evaluates package queries an order of magnitude faster than DIRECT; (2) SKETCHREFINE scales up to sizes that DIRECT cannot handle directly; (3) SKETCHREFINE produces packages of high quality (similar objective value as the packages returned by DIRECT); (4) the performance of SKETCHREFINE is robust to partitioning on different sets of attributes as long as a query's attributes are mostly covered. This makes offline partitioning effective for entire query workloads.

5.1 Experimental Setup

Software. We implemented our package evaluation system as a layer on top of a traditional relational DBMS. The data itself resides in the database, and the system interacts with the DBMS via SQL when it needs to perform operations on the data. We use PostgreSQL v9.3.9 for our experiments. The core components of our evaluation module are implemented in Python 2.7. The PaQL parser is generated in C++ from a context-free grammar, using GNU Bison [12]. We represent a package in the relational model as a standard relation with schema equivalent to the schema of the input relation. A package is materialized into the DBMS only when necessary (for example, to compute its objective value).

We employ IBM's CPLEX [16] v12.6.1 as our black-box ILP solver. When the algorithm needs to solve an ILP problem, the corresponding data is retrieved from the DBMS and passed to CPLEX using tuple iterator APIs to avoid having more than one copy of the same data stored in main memory at any time. We used the same settings for all solver executions: we set its working memory to 512MB; we instructed CPLEX to store exceeding data used during the solve procedure on disk in a compressed format, rather than using the operating system's virtual memory, which, as per the documentation, may degrade the solver's performance; we instructed CPLEX to emphasize optimality versus feasibility to dampen the effect of internal heuristics that the solver may employ on particularly hard problems; we enabled CPLEX's memory emphasis parameter, which instructs the solver to conserve memory where possible; we set a solving time limit of one hour; we also made sure that the operating system would kill the solver process whenever it uses the entire available main memory.

TPC-H query	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Max # of tuples	6M	6M	6M	6M	240k	11.8M	6M

Figure 3: Size of the tables used in the TPC-H benchmark.

Dataset	Dataset size	Size threshold τ	Partitioning time
Galaxy	5.5M tuples	550k tuples	348 sec.
TPC-H	17.5M tuples	1.8M tuples	1672 sec.

Figure 4: Partitioning time for the two datasets, using the workload attributes and with no radius condition.

Environment. We run all experiments on a ProLiant DL160 G6 server equipped with two twelve-core Intel Xeon X5650 CPUs at 2.66GHz each, with 15GB or RAM, with a single 7200 RPM 500GB hard drive, running CentOS release 6.5.

Datasets and queries. We demonstrate the performance of our query evaluation methods using both real-world and benchmark data. The real-world dataset consists of approximately 5.5 million tuples extracted from the Galaxy view of the Sloan Digital Sky Survey (SDSS) [28], data release 12. For the benchmark datasets we used TPC-H [29], with table sizes up to 11.8 million tuples.

For each of the two datasets, we constructed a set of seven package queries, by adapting existing SQL queries originally designed for each of the two datasets. For the Galaxy dataset, we adapted some of the real-world sample SQL queries available directly from the SDSS website.[2] For the TPC-H dataset, we adapted seven of the SQL query templates provided with the benchmark that contained enough numerical attributes. We performed query specification manually, by transforming SQL aggregates into global predicates or objective criteria whenever possible, selection predicates into global predicates, and by adding cardinality bounds. We did not include any base predicates in our package queries because they can always be pre-processed by running a standard SQL query over the input dataset (Section 3), and thus eliminated beforehand. For the Galaxy queries, we synthesized the global constraint bounds by multiplying the original selection constraint bound by the expected size of the feasible packages. For the TPC-H queries, we generated global constraint bounds uniformly at random by multiplying random values in the value range of a specific attribute by the expected size of the feasible packages. The original TPC-H SQL queries involve attributes across different relations and compute various group-by aggregates. In order to transform these queries into single-relation package queries, we processed the original TPC-H tables to produce a single pre-joined table, obtained with full outer joins, containing all attributes needed by all the TPC-H package queries in our benchmark. This table contained approximately 17.5 million tuples. For each TPC-H package query, we then extracted the subset of tuples having non-NULL values on all the query attributes. The size of each resulting table is reported in Figure 3.

Comparisons. We compare DIRECT with SKETCHREFINE. Both methods use the ILP formulation (Section 3) to transform package

[2] http://cas.sdss.org/dr12/en/help/docs/realquery.aspx

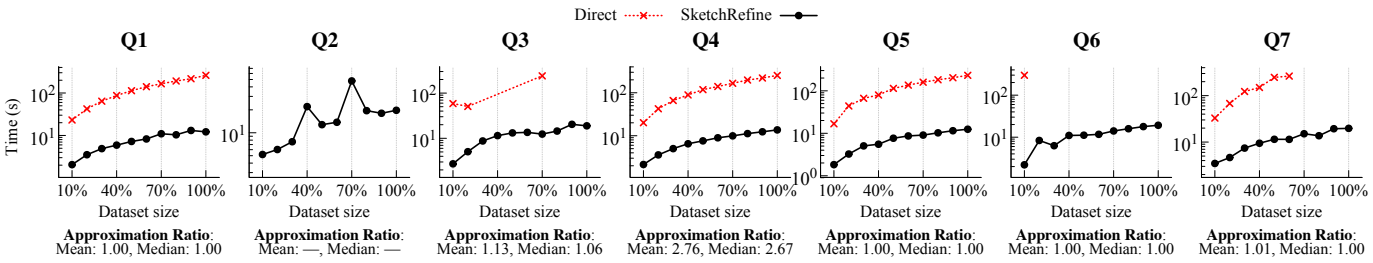

Figure 5: Scalability on the Galaxy benchmark. SKETCHREFINE uses an offline partitioning computed on the full dataset, using the workload attributes, $\tau = 10\%$ of the dataset size, and no radius condition. DIRECT scales up to millions of tuples in about half of the queries, but it fails on the other half. SKETCHREFINE scales up nicely in all cases, and runs about an order of magnitude faster than DIRECT. Its approximation ratio is always low, even though the partitioning is constructed without radius condition.

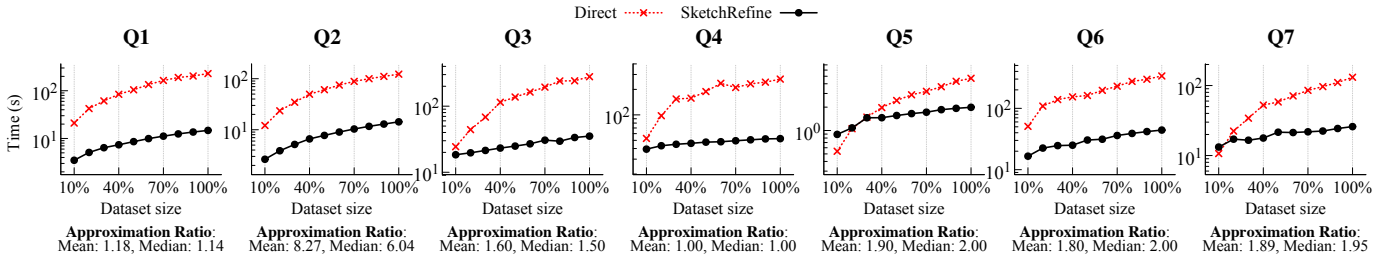

Figure 6: Scalability on the TPC-H benchmark. SKETCHREFINE uses an offline partitioning computed on the full dataset, using the workload attributes, $\tau = 10\%$ of the dataset size, and no radius condition. DIRECT scales up to millions of tuples in all queries. The response time of SKETCHREFINE is about an order of magnitude less than DIRECT, and its approximation ratio is generally very low, even though the partitioning is constructed without radius condition.

queries into ILP problems: DIRECT translates and solves the original query; SKETCHREFINE translates and solves the sub-queries (Section 4), and uses *hybrid sketch query* (Section 4.4) as the only strategy to cope with infeasible initial queries.

Metrics. We evaluate methods on their efficiency and effectiveness.

Response time: We measure response time as wall-clock time to generate an answer package. This includes the time taken to translate the PaQL query into one or several ILP problems, the time taken to load the problems into the solver, and the time taken by the solver to produce a solution. We exclude the time to materialize the package solution to the database and to compute its objective value.

Approximation ratio: Recall that SKETCHREFINE is always guaranteed to return an approximate answer with respect to DIRECT (Section 4.3). In order to assess the quality of a package returned by SKETCHREFINE, we compare its objective value with the objective value of the package returned by DIRECT on the same query. Using Obj_S and Obj_D to denote the objective values of SKETCHREFINE and DIRECT, respectively, we compute the empirical *approximation ratio* $\frac{Obj_D}{Obj_S}$ for maximization queries, and $\frac{Obj_S}{Obj_D}$ for minimization queries. An approximation ratio of one indicates that SKETCHREFINE produces a solution with same objective value as the solution produced by the solver on the entire problem. Typically, the approximation ratio is greater than or equal to one. However, since the solver employs several approximations and heuristics, values lower than one, which means that SKETCHREFINE produces a better package than DIRECT, are possible in practice.

5.2 Results and Discussion

We evaluate three fundamental aspects of our algorithms: (1) their query response time and approximation ratio with increasing dataset sizes; (2) the impact of varying partitioning size thresholds (τ) on SKETCHREFINE's performance; (3) the impact of the attributes used in offline partitioning on query runtime.

5.2.1 *Query performance as data set size increases*

In our first set of experiments, we evaluate the scalability of our methods on input relations of increasing size. First, we partitioned each dataset using the union of all package query attributes in the workload: we refer to these partitioning attributes as the *workload attributes*. We did not enforce a radius condition (ω) during partitioning for two reasons: (1) to show that an offline partitioning can be used to answer efficiently and effectively both maximization and minimization queries, even though they would normally require different radii; (2) to demonstrate the effectiveness of SKETCHREFINE in practice, even without having theoretical guarantees in place.

We perform offline partitioning setting the partition size threshold τ to 10% of the dataset size. Figure 4 reports the partitioning times for the two datasets. We derive the partitionings for the smaller data sizes (less than 100% of the dataset) in the experiments, by randomly removing tuples from the original partitions. This operation is guaranteed to maintain the size condition.

Figures 5 and 6 report our scalability results on the Galaxy and TPC-H benchmarks, respectively. The figures display the query runtimes in seconds on a logarithmic scale, averaged across 10 runs for each datapoint. At the bottom of each figure, we also report the mean and median approximation ratios across all dataset sizes. The graph for Q2 on the galaxy dataset does not report approximation ratios, because DIRECT evaluation fails to produce a solution for this query across all data sizes. We observe that DIRECT can scale up to millions of tuples in three of the seven Galaxy queries, and in all of the TPC-H queries. Its run-time performance degrades, as expected, when data size increases, but even for very large datasets DIRECT is usually able to answer the package queries in less than

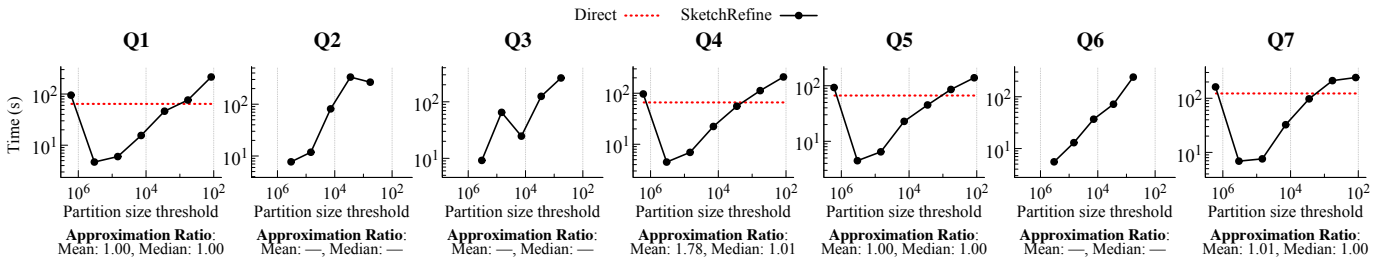

Figure 7: Impact of partition size threshold τ on the Galaxy benchmark, using 30% of the original dataset. Partitioning is performed at each value of τ using all the workload attributes, and with no radius condition. The baseline DIRECT and the approximation ratios are only shown when DIRECT is successful. The results show that τ has a major impact on the running time of SKETCHREFINE, but almost no impact on the approximation ratio. DIRECT can be an order of magnitude faster than DIRECT with proper tuning of τ.

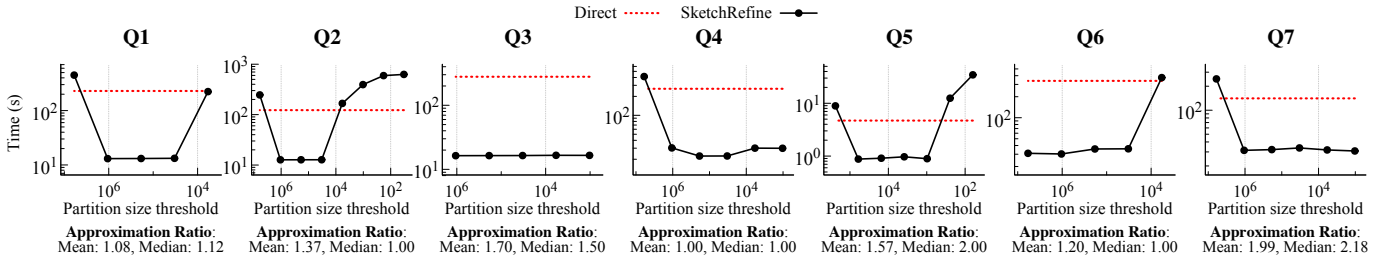

Figure 8: Impact of partition size threshold τ on the TPC-H benchmark, using the full datasets. Partitioning is performed at each value of τ using all the workload attributes, and with no radius condition. The baseline DIRECT and the approximation ratios are only shown when DIRECT is successful. The results show that τ has a major impact on the running time of SKETCHREFINE, but almost no impact on the approximation ratio. DIRECT can be an order of magnitude faster than DIRECT with proper tuning of τ.

a few minutes. However, DIRECT has high failure rate for some of the Galaxy queries, indicated by the missing data points in some graphs (queries Q2, Q3, Q6 and Q7 in Figure 5). This happens when CPLEX uses the entire available main memory while solving the corresponding ILP problems. For some queries, such as Q3 and Q7, this occurs with bigger dataset sizes. However, for queries Q2 and Q6, DIRECT even fails on small data. This is a clear demonstration of one of the major limitations of ILP solvers: they can fail even when the dataset can fit in main memory, due to the complexity of the integer problem. In contrast, our scalable SKETCHREFINE algorithm is able to perform well on all dataset sizes and across all queries. SKETCHREFINE consistently performs about an order of magnitude faster than DIRECT across all queries, both on real-world data and benchmark data. Its running time is consistently below one or two minutes, even when constructing packages from millions of tuples.

Both the mean and median approximation ratios are very low, usually all close to one or two. This shows that the substantial gain in running time of SKETCHREFINE over DIRECT does not compromise the quality of the resulting packages. Our results indicate that the overhead of partitioning with a radius condition is often unnecessary in practice. Since the approximation ratio is not enforced, SKETCHREFINE can potentially produce bad solutions, but this happens rarely. In our experiments, this only occurred with query Q2 from the TPC-H benchmark.

5.2.2 *Effect of varying partition size threshold*

The size of each partition, controlled by the partition size threshold τ, is an important factor that can impact the performance of SKETCHREFINE: Larger partitions imply fewer but larger subproblems, and smaller partitions imply more but smaller subproblems. Both cases can significantly impact the performance of SKETCHREFINE. In our second set of experiments, we vary τ, which is used during partitioning to enforce the size condition (Section 4.1), to study its effects on the query response time and the approximation ratio of SKETCHREFINE. In all cases, along the lines of the previous experiments, we do not enforce a radius condition. Figures 7 and 8 show the results obtained on the Galaxy and TPC-H benchmarks, using 30% and 100% of the original data, respectively. We vary τ from higher values corresponding to fewer but larger partitions, on the left-hand size of the x-axis, to lower values, corresponding to more but smaller partitions. When DIRECT is able to produce a solution, we also report its running time (horizontal line) as a baseline for comparison.

Our results show that the partition size threshold has a major impact on the execution time of SKETCHREFINE, with extreme values of τ (either too low or too high) often resulting in slower running times than DIRECT. With bigger partitions, on the left-hand side of the x-axis, SKETCHREFINE takes about the same time as DIRECT because both algorithms solve problems of comparable size. When the size of each partition starts to decrease, moving from left to right on the x-axis, the response time of SKETCHREFINE decreases rapidly, reaching about an order of magnitude improvement with respect to DIRECT. Most of the queries show that there is a "sweet spot" at which the response time is the lowest: when all partitions are small, and there are not too many of them. The point is consistent across different queries, showing that it only depends on the input data size (refer to Figure 3 for the different TPC-H data sizes). After that point, although the partitions become smaller, the number of partitions starts to increase significantly. This increase has two negative effects: it increases the number of representative tuples, and thus the size and complexity of the initial sketch query, and it increases the number of groups that REFINE may need to refine to construct the final package. This causes the running time of SKETCHREFINE, on the right-hand side of the x-axis, to increase

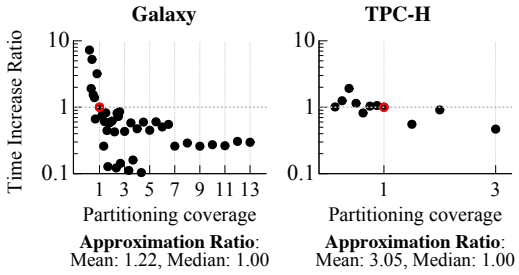

Figure 9: Increase or decrease ratio in running time of SKETCHREFINE with different partitioning coverages. Coverage one, shown by the red dot, is obtained by partitioning on the query attributes. The results show an improvement in running time when partitioning is performed on supersets of the query attributes, with very good approximation ratios.

again and reach or surpass the running time of DIRECT. The mean and median approximation ratios are in all cases very close to one, indicating that SKETCHREFINE retains very good quality regardless of the partition size threshold.

5.2.3 *Effect of varying partitioning coverage*

In our final set of experiments, we study the impact of offline partitioning on the query response time and the approximation ratio of SKETCHREFINE. We define the *partitioning coverage* as the ratio between the number of partitioning attributes and the number of query attributes. For each query, we test partitionings created using: (a) exactly the query attributes (coverage = 1), (b) proper subsets of the query attributes (coverage < 1), and (c) proper supersets of the query attributes (coverage > 1).

For each query, we report the effect of the partitioning coverage on query runtime as the ratio of a query response time over the same query's response time when coverage is one: a higher ratio (> 1) indicates slower response time and a lower ratio (< 1) indicates a faster response time. Figure 9 reports the results on the Galaxy and the TPC-H datasets. The Galaxy dataset has many more numerical attributes than the TPC-H dataset, allowing us to experiment with higher values of coverage. The response time of SKETCHREFINE improves on both datasets when the offline partitioning covers a superset of the query attributes, whereas it tends to increase when it only considers a subset of the query attributes. The mean and median approximation ratios are consistently low, indicating that the quality of the packages returned by SKETCHREFINE remains unaffected by the partitioning coverage. These results demonstrate that SKETCHREFINE is robust to imperfect partitionings, which do not cater precisely to the query attributes. Moreover, using a partitioning over a superset of a query's attributes typically leads to better performance. This means that partitioning can be performed offline using the union of the attributes of an anticipated workload, or even using all the attributes of a relation.

6. RELATED WORK

Package recommendations. Package or set-based recommendation systems are closely related to package queries. A package recommendation system presents users with interesting sets of items that satisfy some global conditions. These systems are usually driven by specific application scenarios. For instance, in the CourseRank [25] system, the items to be recommended are university *courses*, and the types of constraints are course-specific (e.g., prerequisites, incompatibilities, etc.). *Satellite packages* [1] are sets of items, such as smartphone accessories, that are compatible with a "central" item, such as a smartphone. Other related problems in the area of package recommendations are *team formation* [21, 2], and recommendation of *vacation* and *travel packages* [8]. Queries expressible in these frameworks are also expressible in PaQL, but the opposite does not hold. The complexity of set-based package recommendation problems is studied in [9], where the authors show that computing top-k packages with a conjunctive query language is harder than NP-complete.

Semantic window queries and Searchlight. Packages are also related to *semantic windows* [17]. A semantic window defines a contiguous subset of a grid-partitioned space with certain global properties. For instance, astronomers can partition the night sky into a grid, and look for regions of the sky whose overal brightness is above a specific threshold. If the grid cells are precomputed and stored into an input relation, these queries can be expressed in PaQL by adding a global constraint (besides the brightness requirement) that ensures that all cells in a package must form a contiguous region in the grid space. Packages, however, are more general than semantic windows because they allow regions to be non-contiguous, or to contain gaps. Moreover, package queries also allow optimization criteria, which are not expressible in semantic window queries.

A recent extension to methods for answering semantic window queries is Searchlight [18], which expresses these queries in the form of constraint programs. Searchlight uses in-memory synopses to quickly estimate aggregate values of contiguous regions. However, it does not support synopses for non-contiguous regions, and thus it cannot solve arbitrary package queries. Searchlight has several other major differences with our work: (1) it computes optimal solutions by enumerating the feasible ones and retaining the optimal, whereas our methods do not require enumeration; (2) Searchlight assumes that the solver implements redundant and arbitrary data access paths while solving the problems, whereas our approach decouples data access from the solving procedure; (3) Searchlight does not provide a declarative query language such as PaQL; (4) unlike SKETCHREFINE, Searchlight does not allow solvers to scale up to a very large number of variables. At the time of this submission, Searchlight has not been made available by the authors and thus we could not run a comparison for the types of queries that it can express.

How-to queries. Package queries are related to how-to queries [22], as they both use an ILP formulation to translate the original queries. However, there are several major differences between package queries and how-to queries: package queries specify tuple collections, whereas how-to queries specify updates to underlying datasets; package queries allow a tuple to appear multiple times in a package result, while how-to queries do not model repetitions; PaQL is SQL-based whereas how-to queries use a variant of Datalog; PaQL supports arbitrary Boolean formulas in the SUCH THAT clause, whereas how-to queries can only express conjunctive conditions.

Constraint query languages. The principal idea of constraint query languages (CQL) [19] is that a tuple can be generalized as a conjunction of constraints over variables. This principle is very general and creates connections between declarative database languages and constraint programming. However, prior work focused on expressing constraints over tuple values, rather than over sets of tuples. In this light, PaQL follows a similar approach to CQL by embedding in a declarative query language methods that handle higher-order constraints. However, our package query engine design allows for the direct use of ILP solvers as black box components, automatically transforming problems and solutions from one domain to the other. In contrast, CQL needs to appropriately adapt the algorithms themselves between the two domains, and existing literature does not provide this adaptation for the constraint types in PaQL.

ILP approximations. There exists a large body of research in approximation algorithms for problems that can be modeled as integer linear programs. A typical approach is *linear programming relaxation* [30] in which the integrality constraints are dropped and variables are free to take on real values. These methods are usually coupled with *rounding* techniques that transform the real solutions to integer solutions with provable approximation bounds. None of these methods, however, can solve package queries on a large scale because they all assume that the LP solver is used on the entire problem. Another common approach to approximate a solution to an ILP problem is the *primal-dual method* [13]. All primal-dual algorithms, however, need to keep track of all primal and dual variables and the coefficient matrix, which means that none of these methods can be employed on large datasets. On the other hand, rounding techniques and primal-dual algorithms could potentially benefit from the SKETCHREFINE algorithm to break down their complexity on very large datasets.

Approximations to subclasses of package queries. Like package queries, *optimization under parametric aggregation constraints* (OPAC) queries [14] can construct sets of tuples that collectively satisfy summation constraints. However, existing solutions to OPAC queries have several shortcomings: (1) they do not handle tuple repetitions; (2) they only address *multi-attribute knapsack queries*, a subclass of package queries where all global constraints are of the form $\mathsf{SUM}() \leq c$, with a $\mathsf{MAXIMIZE\ SUM}()$ objective criterion; (3) they may return infeasible packages; (4) they are conceptually different from SKETCHREFINE, as they generate approximate solutions in a pre-processing step, and packages are simply retrieved at query time using a multi-dimensional index. In contrast, SKETCHREFINE does not require pre-computation of packages. Package queries also encompass *submodular* optimization queries, whose recent approximate solutions use greedy distributed algorithms [23].

7. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a complete system that supports the specification and efficient evaluation of package queries. We presented PaQL, a declarative extension to SQL, and theoretically established its expressiveness, and we developed a flexible approximation method, with strong theoretical guarantees, for the evaluation of PaQL queries on large-scale datasets. Our experiments on real-world and benchmark data demonstrate that our scalable evaluation strategy is effective and efficient over varied data sizes and query workloads, and remains robust under suboptimal conditions and parameter settings.

In our future work, we plan to extend our evaluation methods to larger classes of package queries, including multi-relation and non-linear queries, and we intend to investigate parallelization strategies for SKETCHREFINE. Package queries pose interesting challenges on usability aspects as well: Our goal is to develop interaction and learning methods that let users identify their ideal packages without having to specify a full and precise PaQL query.

Acknowledgements. This material is based upon work supported by the National Science Foundation under grants IIS-1420941, IIS-1421322, and IIS-1453543.

8. REFERENCES

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

[2] A. Baykasoglu, T. Dereli, and S. Das. Project team selection using fuzzy optimization approach. *Cybernetic Systems*, 38(2):155–185, 2007.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] J. Bisschop. *AIMMS Optimization Modeling*. Paragon Decision Technology, 2006.

[5] M. Brucato, J. F. Beltran, A. Abouzied, and A. Meliou. Scalable package queries in relational database systems. *CoRR*, abs/1512.03564, 2015.

[6] M. Brucato, R. Ramakrishna, A. Abouzied, and A. Meliou. PackageBuilder: From tuples to packages. *PVLDB*, 7(13):1593–1596, 2014.

[7] W. Cook and M. Hartmann. On the complexity of branch and cut methods for the traveling salesman problem. *Polyhedral Combinatorics*, 1:75–82, 1990.

[8] M. De Choudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu. Automatic construction of travel itineraries using social breadcrumbs. In *HyperText*, pages 35–44, 2010.

[9] T. Deng, W. Fan, and F. Geerts. On the complexity of package recommendation problems. In *PODS*, pages 261–272, 2012.

[10] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[11] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[12] GNU Bison. `https://www.gnu.org/software/bison/`.

[13] M. X. Goemans and D. P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. *Approximation algorithms for NP-hard problems*, pages 144–191, 1997.

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

[15] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Applied statistics*, pages 100–108, 1979.

[16] IBM CPLEX Optimization Studio. `http://www.ibm.com/software/commerce/optimization/cplex-optimizer/`.

[17] A. Kalinin, U. Cetintemel, and S. Zdonik. Interactive data exploration using semantic windows. In *SIGMOD*, pages 505–516, 2014.

[18] A. Kalinin, U. Çetintemel, and S. B. Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB*, 8(10):1094–1105, 2015.

[19] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 1(51):26–52, 1995.

[20] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009.

[21] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *SIGKDD*, pages 467–476, 2009.

[22] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.

[23] B. Mirzasoleiman, A. Karbasi, R. Sarkar, and A. Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *NIPS*, 2013.

[24] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[25] A. G. Parameswaran, P. Venetis, and H. Garcia-Molina. Recommendation systems with complex constraints: A course recommendation perspective. *ACM TOIS*, 29(4):1–33, 2011.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] F. Pinel and L. R. Varshney. Computational creativity for culinary recipes. In *CHI*, pages 439–442, 2014.

[28] The Sloan Digital Sky Survey. `http://www.sdss.org/`.

[29] The TPC-H Benchmark. `http://www.tpc.org/tpch/`.

[30] D. P. Williamson and D. B. Shmoys. *The design of approximation algorithms*. Cambridge University Press, 2011.