DOI:10.1145/3299881

Scalable Computation of High-Order Optimization Queries

By Matteo Brucato, Azza Abouzied, and Alexandra Meliou

Abstract

Constrained optimization problems are at the heart of significant applications in a broad range of domains, including finance, transportation, manufacturing, and healthcare. Modeling and solving these problems has relied on application-specific solutions, which are often complex, error-prone, and do not generalize. Our goal is to create a domain-independent, declarative approach, supported and powered by the system where the data relevant to these problems typically resides: the database. We present a complete system that supports *package queries*, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets, allowing the declarative specification and efficient evaluation of a significant class of constrained optimization problems—integer linear programs (ILP)—within a database.

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.

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 2000 and 2500 calories in total, and with a low total intake of saturated fats.*

Similar scenarios, requiring complex, high-order constraints arise frequently, and in many practical settings. A broad set of domains have applications that boil down to modeling and solving constrained optimization problems, for example, coordinating fleet and crew assignments in airline scheduling to reduce delays and costs,[19] managing delinquent consumer credit to minimize losses,[14] optimizing organ transplant allocation and acceptance,[1] and planning of cancer radiotherapy treatments.[20, 21] A significant class of constrained optimization problems are *integer linear programs* (ILP). ILP solutions alone account for billions in US dollars of projected benefits within each of these and other industry sectors.[7]

Modeling and solving these problems has relied on application-specific solutions,[2, 9, 13, 17, 23, 18] which can often be complex and error-prone, and fail to generalize. Our goal is to create a domain-independent, declarative approach, supported and powered by the system where the data relevant to these problems typically resides: the database. We present a complete system that supports *package queries*, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets, allowing the declarative specification and efficient evaluation of a significant class of constrained optimization problems—ILP—within a database. 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 reside 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 addresses *three important challenges*. The first challenge is to support *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 should be between 2000 and 2500 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. Our goal is to maintain the declarative power of SQL, while extending its expressiveness to allow for the easy specification of packages.

The second challenge relates to the *evaluation* of package queries. Due to their combinatorial complexity, package queries are harder to evaluate than traditional database queries.[10] Package queries are in fact as hard as ILP.[5] Existing database technology is ineffective at

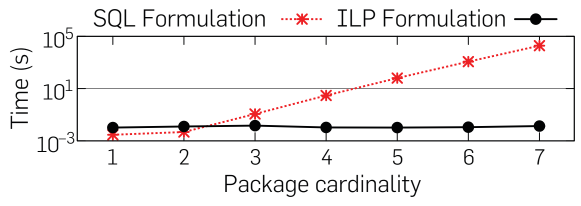
The original version of this paper is entitled "Scalable Package Queries in Relational Database Systems" and was published in the *Proceedings of the VLDB Endowment*, Vol. 9, No. 7 (2016), 576–587.

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. 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 (SDSS) dataset,[22] 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.

The third challenge pertains to query evaluation *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.

Our work addresses these challenges through the design of language and algorithmic support for the specification and evaluation of package queries. We present PaQL (Package Query Language), a declarative language that provides simple extensions to standard SQL to support constraints at the package level. PaQL is at least as expressive as ILP, which implies that evaluation of package queries is NP-hard.[5] 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 ILP. This translation allows for the use of highly-optimized external solvers for the evaluation of package queries. 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 in 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 offers strong approximation guarantees for the package results compared to DIRECT. We present an extensive experimental evaluation on real-world data 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.

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



2. LANGUAGE SUPPORT FOR PACKAGES

Database 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. In the limited case of packages with strict cardinality, that is, a fixed number of tuples, it is possible to express package queries using relational self-joins. The query of Example 1 requires three meals (a package with cardinality three) and can be expressed as a three-way self-join:

```
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
```

Such a 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.

Specifying packages using recursion. 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 declarative and intuitive package specification. In this section, we describe PaQL, a declarative query language that introduces simple extensions to SQL to define package semantics and package-level constraints. Figure 2 shows the general syntax of PaQL (left) and the specification for the query of Example 1 (right), which we use as a running example to demonstrate PaQL's features. Square brackets enclose optional clauses and arguments, and a vertical bar separates syntax alternatives. In this specification, `repeat` is a non-negative integer; `w_expression` is a Boolean expression over tuple values (as in standard SQL) and can only contain references to `relation_name` and `relation_alias`; `st_expression` is a Boolean expression and `obj_expression` is an expression over aggregate functions or SQL subqueries with aggregate functions; both `st_expression` and `obj_expression` can

Figure 2. Specification of the PaQL syntax (left), and the PaQL query for Example 1 (right).

PaQL syntax specification

```
SELECT PACKAGE (*|column_name [,...]) [AS] package_name
FROM relation_name [AS] relation_alias
          [REPEAT repeat] [,...]
[WHERE w_expression ]
[SUCH THAT st_expression ]
[ (MINIMIZE|MAXIMIZE) obj_expression ]
```

PaQL query for Example 1

```
Q: SELECT      PACKAGE (*) 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.sat_fat)
```

only contain references to `package_name`, which specifies the name of the package result.

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

```
Q1: SELECT  *             Q2: SELECT  PACKAGE(*) 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. Similar to SQL, the PaQL syntax allows the specification of the output schema in the SELECT clause. For example, PACKAGE(sat_fat, kcal) only returns the saturated fat and calorie attributes of the package.

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 $\mathcal{Q}$ (Figure 2) that restrict the number of package results.

Repetition constraints. The REPEAT 0 statement in query $\mathcal{Q}$ from Figure 2 specifies that each tuple from the input relation Recipe can appear in a package result at most once (no repetitions are allowed). If this restriction is absent (as in query $\mathcal{Q}_2$), the multiplicity of a tuple is unbounded. 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, REPEAT ρ allows a package to repeat tuples up to ρ times, resulting in $(2 + \rho)^n$ 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 $\mathcal{Q}$ from Figure 2 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, for example, COUNT(P.*) = 3, which limits the query results to packages of exactly 3 tuples.

The global predicates in query $\mathcal{Q}$ abbreviate aggregates that are in reality SQL subqueries. For example, COUNT(P.*) = 3, abbreviates (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, for example, MINIMIZE SUM(P.sat_fat). Similar to global predicates, this form is a shorthand for MINIMIZE (SELECT SUM(sat_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.

Expressiveness and complexity. PaQL can express general ILP, which means that evaluation of package queries is NP-complete.[4, 5] As a first step in package evaluation, we proceed to show how a PaQL query can be transformed into a linear program and solved using general ILP solvers.

3. ILP FORMULATION

In this section, we present an ILP formulation for package queries, which is at the core of our evaluation methods DIRECT and SKETCHREFINE. The results in this section are inspired by the translation rules employed by Tiresias[15] to answer *how-to queries*.

3.1. PaQL to ILP translation

Let R indicate the input relation of the package query, $n = |\mathsf{R}|$ be 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.

Repetition constraint. The REPEAT keyword, expressible in the FROM clause, restricts the domain that the variables can take on. Specifically, REPEAT ρ implies $0 \leq x_i \leq \rho + 1$.

Base predicate. Let β be a base predicate, for example, 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 $t_i \notin R_\beta$.

Global predicate. Each global predicate in the SUCH THAT clause takes the form $f(P) \odot v$. For each such predicate, we derive a linear function $f'(\overline{x})$ over the integer variables. A cardinality constraint $f(P) = \text{COUNT}(P.*)$ is translated into a linear function $f'(\overline{x}) = \sum_i x_i$. A summation constraint $f(P) = \text{SUM}(P.\text{attr})$ is translated into a linear function $f'(\overline{x}) = \sum_i (t_i.\text{attr}) x_i$. Other nontrivial constraints and 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.[3] We refer to the original version of this paper for further details.[4, 5]

Objective clause. We encode MAXIMIZE $f(P)$ as max $f'(\overline{x})$, where $f'(\overline{x})$ is the encoding of $f(P)$. Similarly MINIMIZE $f(P)$ is encoded as min $f'(\overline{x})$.

EXAMPLE 2 (ILP TRANSLATION). *Figure 3 shows a toy example of the* Recipes *table, with two columns and 5 tuples. To transform* $\mathcal{Q}$ *into an ILP, we first create a non-negative, integer variable for each tuple:* $x_1, \ldots, x_5$. *The cardinality constraint specifies that the sum of the* x_i *variables should be exactly 3. The global constraint on* SUM(P.kcal) *is formed by multiplying each* x_i *with the value of the* kcal *column of the corresponding tuple, and specifying that the sum should be between 2 and 2.5. The objective of minimizing* SUM(P.sat_fat) *is similarly formed by multiplying each* x_i *with the* sat_fat *value of the corresponding tuple.*

3.2. Query evaluation with DIRECT

Using the ILP formulation, we develop DIRECT, our basic evaluation method for package queries. In Section 4, we extend this technique to our main algorithm, SKETCHREFINE, which supports efficient package evaluation in large datasets. Package evaluation with DIRECT employs three steps:

1. **Base Relations:** We first compute the base relations, such as R_β, R_c, and R_p, with a series of standard SQL queries, one for each, or by simply scanning R once and populating these relations simultaneously.
2. **ILP Formulation:** We transform the PaQL query to an ILP problem using the rules described in Section 3.1. 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.
3. **ILP Execution:** We employ an off-the-shelf ILP solver, as a black box, to get a solution to each of the integer variables x_i. Each x_i informs the number of times tuple t_i should be included in the answer package.

EXAMPLE 3 (ILP SOLUTION). *The ILP solver operating on the program of Figure 3 returns the variable assignments to* x_i *that lead to the optimal solution;* $x_i = 0$ *means that tuple* t_i *is not included in the output package, and* $x_i = k$ *means that tuple* t_i *is included k times. Thus, the result of* $\mathcal{Q}$ *is the package:* $\{t_2, t_3, t_5\}$.

Figure 3. Example ILP formulation and solution for query Q, on a sample Recipe dataset. There are only two packages that satisfy all the constraints, namely $\{t_2, t_3, t_5\}$ and $\{t_1, t_2, t_5\}$, but the first one is the optimal because it minimizes the objective function.

Recipes

	sat_fat	kcal	
t_1	7.1	0.45	$x_1 = 0$
t_2	5.2	0.55	$x_2 = 1$
t_3	3.2	0.25	$x_3 = 1$
t_4	6.5	0.15	$x_4 = 0$
t_5	2.0	1.20	$x_5 = 1$

$$\begin{aligned} \text{min} \quad & 7.1x_1 + 5.2x_2 + 3.2x_3 + 6.5x_4 + 2.0x_5 \\ \text{s.t.} \quad & x_1 + x_2 + x_3 + x_4 + x_5 = 3 \\ & 0.45x_1 + 0.55x_2 + 0.25x_3 \\ & \qquad + 0.15x_4 + 1.20x_5 \geq 2.0 \\ & 0.45x_1 + 0.55x_2 + 0.25x_3 \\ & \qquad + 0.15x_4 + 1.20x_5 \leq 2.5 \\ & x_1, x_2, x_3, x_4, x_5 \in \{0, 1\} \end{aligned}$$

4. SCALABLE PACKAGE EVALUATION

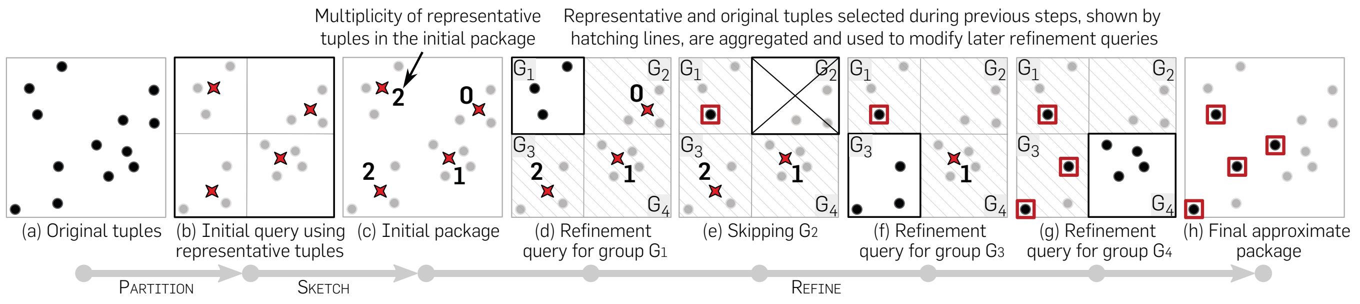
The DIRECT algorithm has two crucial drawbacks. First, it is only applicable if the input relation is small enough to fit entirely in main memory: ILP solvers, such as IBM's CPLEX, require the entire problem to be loaded in memory before execution. Second, even for problems that fit in main memory, this approach may fail due to the complexity of the integer problem. In fact, ILP is a notoriously hard problem, and modern ILP solvers use algorithms, such as *branch-and-cut*,[16] that often perform well in practice, but can "choke" even on small problem sizes due to their exponential worst-case complexity.[8] This may result in unreasonable performance if the solvers use too many resources (main memory, virtual memory, CPU time), eventually thrashing the entire system.

In this section, we present SKETCHREFINE, an approximate divide-and-conquer evaluation technique for efficiently answering package queries on large datasets. Rather than solving the original large problem with DIRECT, 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 DIRECT.

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 4 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, with a representative tuple chosen for each group. This partitioning is performed offline (not at query time).
2. **Sketch (Section 4.2.1):** SKETCHREFINE sketches an

Figure 4. 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 is used while refining the solution for the remaining groups (f and g). The final approximate package (h) contains only original tuples.


Multiplicity of representative tuples in the initial package
Representative and original tuples selected during previous steps, shown by hatching lines, are aggregated and used to modify later refinement queries
G1
G2
G3
G4
0
1
2
(a) Original tuples
(b) Initial query using representative tuples
(c) Initial package
(d) Refinement query for group G1
(e) Skipping G2
(f) Refinement query for group G3
(g) Refinement query for group G4
(h) Final approximate package
PARTITION
SKETCH
REFINE

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, that is, packages that satisfy all the query constraints, but with a possibly sub-optimal objective value that is guaranteed to be within certain approximation bounds. 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. We formalize these properties in Section 4.3.

In the subsequent discussion, we use $\mathsf{R}(\mathsf{attr}_1, \ldots, \mathsf{attr}_k)$ to denote an input relation with k attributes. R is partitioned into m groups $G_1, \ldots, G_m$. Each group $G_i \subseteq \mathsf{R}$, $1 \le i \le m$, has a representative tuple $\tilde{t}_i$, which may not always appear in R. We denote the partitioned space with $\mathcal{P} = \{(G_i, \tilde{t}_i) | 1 \le i \le m\}$. We refer to packages that contain representative tuples as *sketch packages* and packages with only original tuples as *complete packages* (or simply *packages*). 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 $p_{\mathcal{S}}$ to 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 *partitioning attributes* from the input relation R, a *size* threshold, and a set of *diameter* bounds. The size threshold τ, $1 \le \tau \le n$, restricts the size of each partitioning group G_i, $1 \le i \le m$, to a maximum of τ original tuples, that is, $|G_i| \le \tau$. The *diameter* $d_{ij} \ge 0$ of a group G_i, $1 \le i \le m$, on attribute attr_j, $1 \le j \le k$, is the greatest absolute distance between all pairs of tuples within group G_i. The *diameter bounds*, $\omega_{ij} \ge 0$, $1 \le i \le m$, $1 \le j \le k$, require all diameters to be bounded by $d_{ij} \le \omega_{ij}$.

Setting the partitioning parameters. The size threshold, τ, affects the number of partitions, m: a lower τ leads to smaller partitions, but more of them (larger m). 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 lead to an order of magnitude improvement in query response time.

The diameter bounds, ω_{ij}, are not required, but they can be enforced to ensure a desired approximation guarantee. In general, enforcing the diameter limits 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. This is an important trade-off between running time and quality that we also observe in our experiments, and it is a very common characteristic of most approximation schemes.[24]

Partitioning method. Our partitioning procedure 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 diameter limits. First, relation R is augmented with an extra group ID column gid, such that $t.\mathsf{gid} = i$ if tuple t is assigned to group G_i. The procedure initially creates a single group G_1 that includes all the original tuples from relation R, by initializing gid = 1 for all tuples. Our method recursively computes the sizes and diameters of the current groups, as well as the *centroid* of each group. It then partitions the groups that violate either the size or the diameter limits, using the centroids as partitioning boundaries. In the last iteration, the centroids for each group become the representative tuples, $\tilde{t}_i$, $1 \le i \le m$, and get stored in a new *representative relation* $\tilde{\mathsf{R}}(\mathsf{gid}, \mathsf{attr}_1, \ldots, \mathsf{attr}_k)$.

One-time cost. Partitioning is an expensive procedure. Partitioning the data in advance avoids this cost at query time. 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. 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. Note that the same partitioning can be used to support different queries over the same dataset. In our experiments, we show that a single partitioning performs consistently well across different queries.

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 Figure 2.

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

```
Q(R̃): SELECT        PACKAGE(*) AS p_S
      FROM          R̃
      WHERE         R̃.gluten = 'free'
      SUCH THAT
        COUNT(p_S.*) = 3 AND
        SUM(p_S.kcal) BETWEEN 2.0 AND 2.5 AND
        (SELECT COUNT(*) FROM p_S WHERE gid = 1) ≤ |G_1|
        AND ...
        (SELECT COUNT(*) FROM p_S WHERE gid = m) ≤ |G_m|
      MINIMIZE   SUM(p_S.sat_fat)
```

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

Since the representative relation $\tilde{R}$ 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. 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 $\mathcal{Q}(\tilde{R})$ is infeasible, in which case SketchRefine reports the original query $\mathcal{Q}$ as infeasible. This may constitute *false infeasibility*, if $\mathcal{Q}$ is actually feasible. In Section 4.3, we show that the probability of false infeasibility is low and bounded.

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_i with representative $\tilde{t}_i$, let $\tilde{p}_i \subseteq p_{\mathcal{S}}$ be the set of representatives picked from G_i (i.e., $\tilde{t}_i$ with possible duplicates). The algorithm proceeds as follows:

- It derives package $\bar{p}_i$ from $p_{\mathcal{S}}$, by eliminating all instances of $\tilde{t}_i$ from $p_{\mathcal{S}}$. That is, $\bar{p}_i = p_{\mathcal{S}} \setminus \tilde{p}_i$. This is a solution to all groups except G_i.
- The algorithm then constructs a *refine query*, $\mathcal{Q}_i(p_{\mathcal{S}})$, which searches for a set of tuples $p_i \subseteq G_i$ to replace the eliminated representatives:

```
Q_i(p_S): SELECT      PACKAGE(*) AS p_i
          FROM        G_i REPEAT 0
          WHERE       G_i.gluten = 'free'
          SUCH THAT
          COUNT(p_i.*) + COUNT( p̄_i.*) = 3 AND
          SUM(p_i.kcal) + SUM( p̄_i.kcal) BETWEEN 2.0 AND 2.5
          MINIMIZE    SUM(p_i.sat_fat)
```

- The algorithm adds the result of $\mathcal{Q}_i(p_{\mathcal{S}})$, p_i, in the current solution, $p_{\mathcal{S}}$. Now, group G_i is *refined* with actual tuples.

In $\mathcal{Q}_i(p_{\mathcal{S}})$, COUNT($\bar{p}_i$.*) and SUM($\bar{p}_i$ .kcal) are values computed directly on $\bar{p}_i$ before the query is formed. They are used to modify the original constraint bounds to account for tuples and representatives already chosen for all the other groups. The global constraints in $\mathcal{Q}_i(p_{\mathcal{S}})$ ensure that the combination of tuples in p_i and $\bar{p}_i$ satisfy the original query $\mathcal{Q}$. Thus, this step produces the new *refined sketch package* $p'_{\mathcal{S}'} = \bar{p}_i \cup p_i$, where $\mathcal{S}' = \mathcal{S} \setminus \{(G_i, \tilde{t}_i)\}$.

Since G_i has at most τ tuples, the ILP problem corresponding to $\mathcal{Q}_i(p_{\mathcal{S}})$ has at most τ variables. This is typically small enough for the black-box ILP solver to solve using the Direct method. Similar to the sketch query, if τ is too large, SketchRefine can evaluate the query recursively: the tuples in group G_i 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*, $\mathcal{Q}_i(p_{\mathcal{S}})$. Backtracking *greedily prioritizes* the infeasible groups. This choice is motivated by a simple heuristic: if the refinement on G_i fails, it is likely due to choices made by previous refinements; therefore, by prioritizing G_i, we reduce the impact of other groups on the feasibility of $\mathcal{Q}_i(p_{\mathcal{S}})$. This heuristic does not affect the approximation guarantees.

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. The algorithm assumes a (initially random) refinement order for all groups in $\mathcal{S}$ and places them in a priority queue. During refinement, this group order can change by prioritizing groups with infeasible refinements.

Runtime 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. Theoretical guarantees

We present two important results on the theoretical guarantees of SketchRefine: (1) it produces packages that closely approximate the objective value of the packages produced by Direct; (2) the probability of false negatives (i.e., queries incorrectly deemed infeasible) is low and bounded. The extended version of this work[4] includes the formal proofs of both results.

For a desired approximation parameter ε, we can derive diameter bounds ω_{ij} for the offline partitioning that guarantee that SketchRefine will produce a package with objective value $(1\pm\varepsilon)$-factor close to the objective value of the solution generated by Direct for the same query.

Theorem 1 (Approximation Bounds). *Let* $\mathsf{R}(\mathsf{attr}_1, \ldots, \mathsf{attr}_k)$ *be a relation with k attributes, and let* $\mathcal{Q}$ *be a feasible package query with a maximization (minimization, resp.) objective over* R. *Let S be an exact solver that produces an answer to* $\mathcal{Q}$ *with optimal objective value OPT. We denote with ALG the objective value of the package returned by* SketchRefine *using S as a black-box solver. For any* $\varepsilon \in [0, 1)$ $(\varepsilon \in [0, \infty)$, *resp.), there exists* $\beta \in [0, 1)$ $(\beta \in [1, \infty)$, *resp.) that depends on* ε, *such that if* R *is partitioned into m groups with diameter limits:*

$$\omega_{ij} = \min_{t\in G_i}\{|1-\beta|\cdot|t.\mathsf{attr}_j|\}, \quad \forall i \in [1,m],\ \forall j \in [1,k] \qquad (1)$$

then $ALG \geq (1-\varepsilon)OPT$ $(ALG \leq (1+\varepsilon)OPT$, *resp.).*

For a feasible query $\mathcal{Q}$, *false infeasibility* may happen in two cases: (1) when the sketch query $\mathcal{Q}(\tilde{\mathsf{R}})$ 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 2 (False-infeasibility Bounds). *For any query* $\mathcal{Q}$ *and any random package* P, *if* P *is feasible for* $\mathcal{Q}$, *then with high probability: (1) the* Sketch *query* $\mathcal{Q}(\tilde{\mathsf{R}})$ *is feasible; (2) all* Refine *queries* $\mathcal{Q}_i(p_{\mathcal{S}})$, $1 \leq i \leq m$, *are feasible. Thus,* SketchRefine *returns a feasible result.*

5. EXPERIMENTAL EVALUATION

This section presents an extensive experimental evaluation of our techniques for package query execution on real-world data. The 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). We have also performed extensive experiments on benchmark data that demonstrate the robustness of SketchRefine under imperfect partitioning and different approximation parameters.[4, 5]

5.1. Experimental setup

We implemented our package evaluation system as a layer on top of PostgreSQL.[a] The system interacts with the DBMS via SQL and uses IBM's CPLEX[12] as the black-box ILP solver. A package is materialized into the DBMS as a relation, only when necessary (e.g., to compute its objective value). The experiments compare Direct with SketchRefine. Both methods use the PaQL to ILP translation presented in Section 3.1: Direct translates and solves the original query; SketchRefine translates and solves the subqueries. We demonstrate the performance of our query evaluation methods using a real-world dataset consisting of approximately 5.5 million tuples extracted from the Galaxy view of the SDSS,[22] and a workload of seven feasible package queries (Figure 5) constructed by adapting some of the real-world sample SQL queries available directly from the SDSS Website. The experiments use the following efficiency and effectiveness metrics:

Response time. We measure response time as wall-clock time to generate an answer package. This includes the time to translate the PaQL query into one or several ILP problems, the time to load the problems to the solver, and the time the solver takes to produce a solution.

Approximation ratio. We compare the objective value of a package returned by SketchRefine 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 report 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. The higher the approximation ratio, the lower the quality of the result package.

5.2. Results and discussion

We evaluate two fundamental aspects of our algorithms: (1)

[a] Our code is publicly available on our project Website: http://packagebuilder.cs.umass.edu.

Figure 5. Summary of queries in the Galaxy workload. The full PaQL queries appear in the extended version of this work.[4]

Query	$\mathcal{Q}_1$	$\mathcal{Q}_2$	$\mathcal{Q}_3$	$\mathcal{Q}_4$	$\mathcal{Q}_5$	$\mathcal{Q}_6$	$\mathcal{Q}_7$
Objective	max	min	min	min	min	min	max
# of SUM constraints	2	4	2	1	1	5	5
COUNT (*)	BETWEEN 5 AND 10						

their query response time and approximation ratio with increasing dataset sizes; (2) the impact of varying partitioning size thresholds, τ, on SketchRefine's performance.

Query performance as dataset size increases. The first set of experiments evaluates the scalability of our methods on input relations of increasing size. First, we partition each dataset using the union of all package query attributes in the workload: we refer to these partitioning attributes as the *workload attributes*. We do not enforce diameter conditions, ω_{ij}, during partitioning for three reasons: (1) because the diameter conditions may affect the size of the resulting partitions, and we want to tightly control the partition size through the parameter τ; (2) 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 diameters; (3) to demonstrate the effectiveness of SketchRefine in practice, even without having theoretical guarantees in place.

We perform offline partitioning with partition size threshold τ set to 10% of the dataset size. We derive the partitionings for the smaller data sizes (less than 100% of the dataset), by randomly removing tuples from the original partitions. This operation is guaranteed to maintain the size condition.

Figure 6 reports our scalability results on the Galaxy workload. The figure displays the query response time in seconds on a logarithmic scale, averaged across 10 runs for each datapoint. At the bottom of each plot, we also report the mean and median approximation ratios across all dataset sizes. The graph for Q2 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 queries. Its runtime 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 a few minutes. However, Direct has high failure rate for some of the queries, indicated by the missing data points in some graphs (queries Q2, Q3, Q6, and Q7). 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. 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 diameter limits is often unnecessary in practice. Since the approximation ratio is not enforced, SketchRefine can potentially produce bad solutions, but this happens rarely.

Effect of varying partition size threshold. In the second set of experiments, we vary τ, which is used during

Figure 6. Scalability on the Galaxy workload. SketchRefine uses an offline partitioning computed on the full dataset, using the workload attributes, t = 10% of the dataset size, and no diameter condition. Direct scales up to millions of tuples in about half of the queries, but it fails on the other half. SketchRefine scales well 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 diameter conditions.

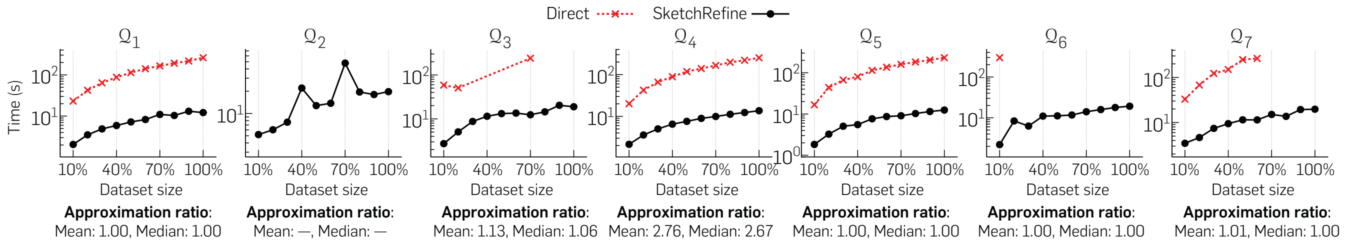
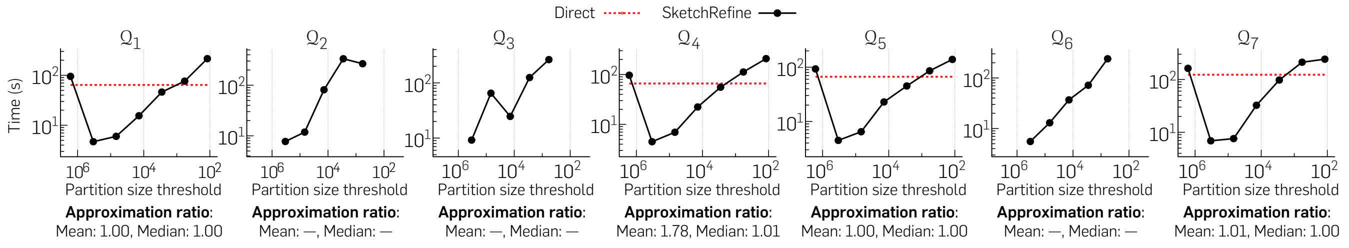


Figure 7. Impact of partition size threshold t on the Galaxy workload, using 30% of the original dataset. Partitioning is performed at each value of t using all the workload attributes, and with no diameter condition. The baseline Direct and the approximation ratios are only shown when Direct is successful. The results show that t 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 t.



partitioning to limit the size of each partition, 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 diameter conditions. Figure 7 show the results obtained on the Galaxy workload, using 30% of the original data. 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.

The 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. This point is consistent across different queries, showing that it only depends on the input data size. 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 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.

6. CONCLUSION AND FUTURE WORK

We introduced a complete system that supports the declarative specification and efficient evaluation of package queries. We presented PaQL, a declarative extension to SQL, 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 data demonstrate that our scalable evaluation strategy is effective and efficient over varied data sizes and queries. We have further extended our techniques and experimental evaluation and placed our research in the context of related work.[4]

Our work so far focused on *deterministic* package queries, but many applications of constrained optimization require support for uncertainty: airline fleet scheduling has uncertain passenger demands, or investment portfolio optimization deals with uncertain returns and risks, etc. We are currently working on extending our system to support optimization of the expected value of an objective function subject to *expectation constraints* of the form $\mathsf{E}(\mathsf{SUM}(x)) \geq b$, or *probabilistic constraints* of the form $\mathsf{SUM}(x) \geq b$ WITH PROBABILITY $\geq 95\%$. The challenge is to ensure robust optimal solutions, computed efficiently, that behave well under the many possible realizations of the uncertain data.

Another open problem is to efficiently handle *incremental* package queries to enable user-facing, interactive constrained optimization applications such as vacation planning. Rather than calling the solver for each incremental query variation from scratch, we are exploring the use of efficient database techniques, such as top-k querying, to provide faster, albeit approximate, solutions for interactive applications.

Acknowledgments

This research is supported by the National Science Foundation under grants IIS-1420941, IIS-1421322, and IIS-1453543.

References

1. Alagoz, O., Schaefer, A.J., Roberts, M.S. *Optimizing Organ Allocation and Acceptance*. Springer, Boston, MA, 2009, 1–24.
2. Baykasoglu, A., Dereli, T., Das, S. Project team selection using fuzzy optimization approach. *Cybern. Syst.* 38, 2 (2007), 155–185.
3. Bisschop, J. *AIMMS Optimization Modeling*. Paragon Decision Technology, 2006.
4. Brucato, M., Abouzied, A., Meliou, A. Package queries: efficient and scalable computation of high-order constraints. *VLDB J.* (Oct. 2017).
5. Brucato, M., Beltran, J.F., Abouzied, A., Meliou, A. Scalable package queries in relational database systems. *PVLDB* 9, 7 (2016), 576–587.
6. Brucato, M., Ramakrishna, R., Abouzied, A., Meliou, A. PackageBuilder: From tuples to packages. *PVLDB* 7, 13 (2014), 1593–1596.
7. Chen, D.-S., Batson, R.G., Dang, Y. *Applied Integer Programming: Modeling and Solution*. John Wiley & Sons, 2011.
8. Cook, W., Hartmann, M. On the complexity of branch and cut methods for the traveling salesman problem. *Polyhedral Comb.* 1 (1990), 75–82.
9. De Choudhury, M., Feldman, M., Amer-Yahia, S., Golbandi, N., Lempel, R., Yu, C. Automatic construction of travel itineraries using social breadcrumbs. In *Proceedings of the 21st ACM Conference on Hypertext and Hypermedia* (Toronto, Ontario, Canada, June 13–16, 2010) ACM, NY, 35–44.
10. Deng, T., Fan, W., Geerts, F. On the complexity of package recommendation problems. In *PODS '12 Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Scottsdale, Arizona, USA, May 21–23, 2012) ACM, NY, 261–272.
11. Finkel, R.A., Bentley, J.L. Quad trees a data structure for retrieval on composite keys. *Acta Inf.* 4, 1 (1974), 1–9.
12. IBM CPLEX Optimization Studio. http://www.ibm.com/software/commerce/optimization/cplex-optimizer/.
13. Lappas, T., Liu, K., Terzi, E. Finding a team of experts in social networks. In *KDD '09 Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France, June 28–July 01, 2009) ACM, NY, 467–476.
14. Makuch, W.M., Dodge, J.L., Ecker, J.G., Granfors, D.C., Hahn, G.J. Managing consumer credit delinquency in the us economy: A multi-billion dollar management science application. *Interfaces* 22, 1 (1992), 90–109.
15. Meliou, A., Suciu, D. Tiresias: The database oracle for how-to queries. In *SIGMOD '12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA, May 20–24, 2012) ACM, NY, 337–348.
16. Padberg, M., Rinaldi, G. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.* 33, 1 (1991), 60–100.
17. Parameswaran, A.G., Venetis, P., Garcia-Molina, H. Recommendation systems with complex constraints: A course recommendation perspective. *ACM TOIS* 29, 4 (2011), 1–33.
18. Pinel, F., Varshney, L.R. Computational creativity for culinary recipes. In *CHI EA '14 CHI '14 Extended Abstracts on Human Factors in Computing Systems* (Toronto, Ontario, Canada, April 26–May 01, 2014) ACM, NY, 439–442.
19. Rushmeier, R.A., Kontogiorgis, S.A. Advances in the optimization of airline fleet assignment. *Transp. Sci.* 31, 2 (1997), 159–169.
20. Sauer, O.A., Shepard, D.M., Mackie, T.R. Application of constrained optimization to radiotherapy planning. *Med. Phys.* 26, 11 (1999), 2359–2366.
21. Terrer, J.M.A., Benede, M.A.N., del Rio, E.B., Llanas, S.C. A feasible application of constrained optimization in the IMRT system. *IEEE Trans. Biomed. Eng.* 54, 3 (2007), 370–379.
22. The Sloan Digital Sky Survey. http://www.sdss.org/.
23. Wang, X., Dong, X.L., Meliou, A. In *SIGMOD '15 Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia, May 31–June 04, 2015) ACM, NY, 1231–1245.
24. Williamson, D.P., Shmoys, D.B. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

Matteo Brucato and Alexandra Meliou ({matteo,ameli}@cs.umass.edu), College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, USA.

Azza Abouzied (azza@nyu.edu), Computer Science, New York University, Abu Dhabi, UAE.