A Scalable Execution Engine for Package Queries

Matteo Brucato[UM]   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
azza@nyu.edu

ABSTRACT

Many modern applications and real-world problems involve the design of item collections, or *packages*: from planning your daily meals all the way to mapping the universe. Despite the pervasive need for packages, traditional data management does not offer support for their definition and computation. This is because traditional database queries follow a powerful, but very simple model: a query defines constraints that each tuple in the result must satisfy. However, a system tasked with the design of packages cannot consider items independently; rather, the system needs to determine if a set of items *collectively* satisfy given criteria.

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. Second, we present a fundamental strategy for evaluating package queries that combines the capabilities of databases and constraint optimization solvers. 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, an efficient and scalable algorithm for package evaluation, which offers strong approximation guarantees. Finally, we present extensive experiments over real-world data. Our 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.

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 [13].*

In these examples, some conditions can be verified on individual 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 [20], team formation [2, 16], vacation and travel planning [7], and computational creativity [21]. 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, 7, 16, 20, 23].

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 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 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. Our goal is to maintain the declarative power of SQL, while extending its expressiveness to allow for the easy specification of packages.
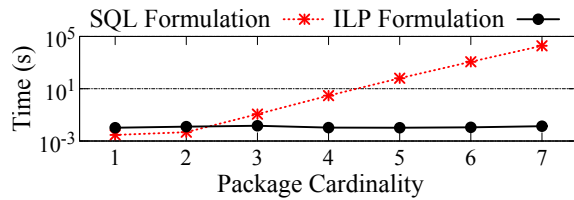
Figure 1: Traditional database technology is ineffective at package evaluation, and the runtime of a SQL formulation of a package query grows exponentially. In contrast, tools such as ILP solvers are more effective.

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 [8]. Package queries are in fact as hard as integer linear programs [4]. 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. 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 [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 integer linear programming, which implies that evaluation of package queries is NP-hard [4]. 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 solvers for the evaluation of package queries. We introduce an off-line 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 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.

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 Q_1 and Q_2 are fundamentally different: 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 Q_2: all possible *multisets* of tuples from the input relation. The result of a package query like 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 Q_2 follows the schema of Recipes.

The specification of Q_2 allows for arbitrary repetitions of tuples, thus, there are infinitely many packages that satisfy the query. Although semantically valid, a query like 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 [5], 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 integer linear programs, which means that evaluation of package queries is NP-complete [4]. 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 [17] to answer *how-to queries*.

3.1 PaQL to ILP Translation

Let R indicate the input relation, $n = |R|$ 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:

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

Base predicate. Let β be a base predicate, e.g., R.gluten = 'free', and R_β the *base 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 $f'(\bar{x})$ over the integer variables. A cardinality constraint $f(P) = \text{COUNT}(P.\ast)$ is linearly translated into $f'(\bar{x}) = \sum_i x_i$. A summation constraint $f(P) = \text{SUM}(P.\text{attr})$ is linearly translated into $f'(\bar{x}) = \sum_i (t_i.\text{attr})x_i$. Other non-trivial 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].

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

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.

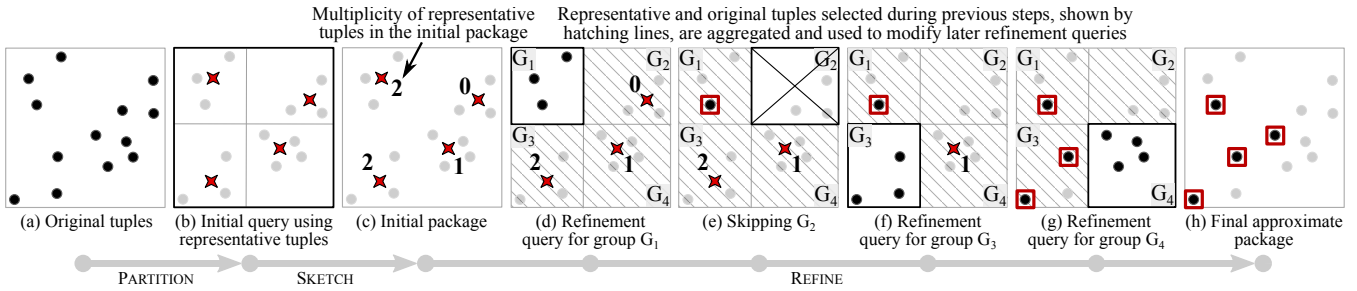Package evaluation with DIRECT employs three simple steps:

1. **ILP formulation.** We transforms a PaQL query to an ILP problem using the rules described in Section 3.1.

2. **Base relation.** We compute the base relation R_β with a traditional SQL query that selects tuples from R that satisfy the base predicate. After this phase, all variables x_i such that $x_i = 0$ can be eliminated from the ILP problem.

3. **ILP execution.** We employ an off-the-shelf ILP solver, as a black box, to get a solution x_i^* 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* [19], that often perform well in practice, but can "choke" even on small problem sizes due to their exponential worst-case complexity [6]. This may result in unreasonable performance due to solvers using too many resources (main memory, virtual memory, CPU time), eventually thrashing the entire system.

4. SCALABLE PACKAGE EVALUATION

In this section, we present SKETCHREFINE, an approximate divide-and-conquer 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 component to answer each individual query. By breaking down the problem into smaller subproblems, the algorithm avoids the drawbacks of the DIRECT approach.

The algorithm is based on an important observation: *similar tuples are likely to be interchangeable within packages*. A group of

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

| (a) Original tuples | (b) Initial query using representative tuples | (c) Initial package | (d) Refinement query for group G_1 | (e) Skipping G_2 | (f) Refinement query for group G_3 | (g) Refinement query for group G_4 | (h) Final approximate package |

PARTITION   SKETCH   REFINE

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.

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). In our implementation, we partition data using *k*-dimensional quad trees [9], 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 *feasible* packages, i.e., packages that satisfy all the query constraints, but with a possibly suboptimal objective value. However, SKETCHREFINE offers strong approximation guarantees compared to the solution generated by DIRECT for the same query. 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 R to denote the input relation of n tuples, $t_i \in R$, $1 \le i \le n$. R is partitioned into m groups $G_1, \ldots, G_m$. Each group G_j, $1 \le j \le m$, has a representative tuple $\tilde{t}_j$, which may not always appear in R. We denote the partitioned space with $\mathcal{P} = \{(G_j, \tilde{t}_j) \mid 1 \le j \le m\}$. We refer to packages that contain some representative tuples as *sketch packages* and packages with only original tuples as *complete packages* (or simply *packages*). 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 (optionally) a *radius limit*. The size threshold τ, $1 \le \tau \le n$, restricts the size of

each partitioning group G_j, $1 \le j \le m$, to a maximum of τ original tuples, i.e., $|G_j| \le \tau$. The *radius* $r_j \ge 0$ 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. The radius limit ω, $\omega \ge 0$, requires that for every partitioning group G_j, $1 \le j \le m$, $r_j \le \omega$.

Setting the partitioning parameters. The size threshold, τ, affects the number of clusters, m, as smaller clusters (lower τ) imply more of them (larger m), especially on skewed datasets. 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 an order of magnitude improvement in query response time.

The *optional* radius limit, ω, helps ensure that a result produced by SKETCHREFINE is within a guaranteed approximation bound from the package that DIRECT would generate. Enforcing a radius limit requires more partitioning iterations, which increases the cost of offline partitioning. However, our experiments show that even without enforcing an approximation guarantee, SKETCHREFINE produces satisfactory answers.

Partitioning method. Our partitioning procedure is based on k-dimensional *quad-tree indexing* [9]. The method recursively partitions a relation into groups until all the groups satisfy the size threshold and meet the radius limit. The procedure initially creates a single group G_1 that includes all the original tuples from relation R. Our method recursively computes the sizes and radii of the current groups, as well as the *centroid* of each group. It then partitions the groups that violate either the size or the radius limits, using the centroids as partitioning boundaries. In the last iteration, the centroids for each group become the representative tuples, $\tilde{t}_j$, $1 \le j \le m$, and get stored in a new *representative relation* $\tilde{R}(\text{gid}, \text{attr}_1, \ldots, \text{attr}_k)$.

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. In order to ensure the approximation guarantees, the partitioning attributes, $\mathcal{A}$, must be a superset of the query attributes. 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 [4]. 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 a multitude of 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 Section 2.1.

4.2.1 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 $\mathcal{Q}$:

$\mathcal{Q}[\tilde{R}]$:
```
SELECT      PACKAGE(R̃) 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.saturated_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_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 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. However, we show that the probability of false infeasibility is low and bounded (Section 4.3).

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(p̄_j.*) = 3 AND
    SUM(p_j.kcal) + SUM(p̄_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. 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.

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.

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 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, and (2) the probability of false negatives (i.e., queries incorrectly deemed infeasible) is low and bounded.

We prove that for a desired approximation parameter ε, we can derive a radius limit ω for the offline partitioning that guarantees 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 1 (APPROXIMATION BOUNDS). *For any feasible package query with a maximization (minimization, resp.) objective and approximation parameter ε, $0 \le \varepsilon < 1$ ($\varepsilon \ge 0$, resp.), any database instance, any set of partitioning attributes $\mathcal{A}$, superset of the numerical query attributes, any size threshold τ, and radius limit:*

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

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.*

For a feasible query $\mathcal{Q}$, *false infeasibility* may happen in two cases: (1) when the sketch query $\mathcal{Q}[\tilde{R}]$ 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 2 (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.*

5. EXPERIMENTAL EVALUATION

We present an extensive experimental evaluation of our techniques for package queries on real-world 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). We have also performed extensive experiments on benchmark data and have investigated the effects of imperfect partitioning over different sets of attributes, demonstrating the robustness of SKETCHREFINE under these variations [4].

5.1 Experimental Setup

We implemented our package evaluation system as a layer on top of PostgreSQL. The system interacts with the DBMS via SQL. A package is materialized into the DBMS, as a relation, only when necessary (for example, to compute its objective value). We employ IBM's CPLEX [12] as our black-box ILP solver. We 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 sub-queries. 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 Sloan Digital Sky Survey (SDSS) [22]. We constructed a set of seven package queries, by adapting some of the real-world sample SQL queries available directly from the SDSS website.

We evaluate methods on their *efficiency* (response time) and *effectiveness* (approximation ratio):

Response time: The wall-clock time to generate an answer package. This only includes the time to translate the PaQL query into one or several ILP problems, the time to load the problems into the solver, and the time taken by the solver 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 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. 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) their query response time and approximation ratio with increasing dataset sizes; (2) the impact of varying partitioning size thresholds (τ) on SKETCHREFINE's performance. Further, our analysis has shown that SKETCHREFINE is robust to imperfect partitioning [4].

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 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 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 with partition size threshold τ set to 10% of the dataset size and without a radius limit. 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 3 reports our scalability results on the Galaxy workload. The figure displays 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 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 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 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 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.

5.2.2 *Effect of varying partition size threshold*

In our second set of experiments, we vary τ, which is used during 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 a radius condition. Figure 4 show the results obtained
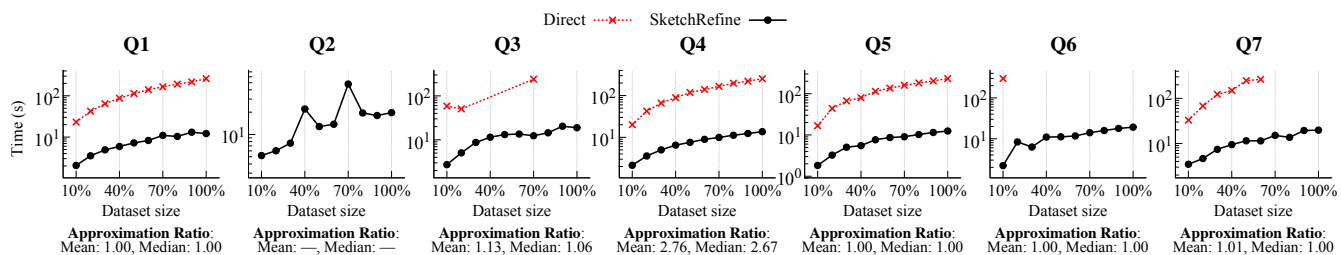
Figure 3: Scalability on the Galaxy workload. SKETCHREFINE uses an offline partitioning computed on the full dataset, using the workload attributes, τ = 10% of the dataset size, and no radius condition. DIRECT scales up to millions of tuples in about half of the queries, but it fails on the other half. 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.
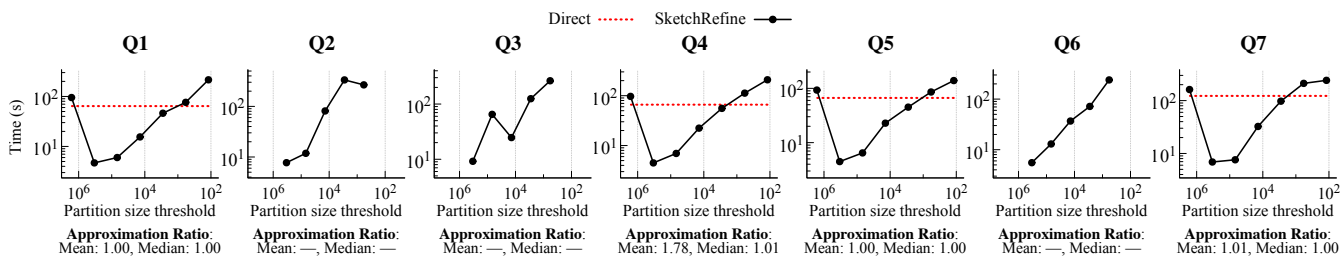
Figure 4: Impact of partition size threshold τ on the Galaxy workload, using 30% of the original dataset. Partitioning is performed at each value of τ using all the workload attributes, and with no radius condition. The baseline DIRECT and the approximation ratios are only shown when DIRECT is successful. 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.

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.

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. 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. RELATED WORK

We discuss related work from the following areas: package recommendation systems, semantic window queries, how-to queries, constraint query languages, and approximation techniques for ILP formulations and subclasses of package queries.

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. Specific application scenarios usually drive these systems. For instance, in the CourseRank [20] 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* [16, 2], and recommendation of *vacation* and *travel packages* [7]. 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 [8], where the authors show that computing top-*k* packages with a conjunctive query language is harder than NP-complete.

Packages are also related to the *semantic windows* [13] expressible in Searchlight [14]. A semantic window defines a contiguous subset of a grid-partitioned space with certain global properties. These queries can be expressed in PaQL by adding global constraints that ensure contiguity in the grid. Packages, however, are more general than semantic windows because they allow regions to be non-contiguous or contain gaps. 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) it 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) it 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.

Package queries are related to how-to queries [17], 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.

The principal idea of constraint query languages (CQL) [15] is that a tuple can be generalized as a conjunction of constraints over variables. This general principle 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. PaQL follows a similar approach to CQL by embedding higher-order constraints in a declarative query language. 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.

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* [24] 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* [10]. 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.

Like package queries, *optimization under parametric aggregation constraints* (OPAC) queries [11] 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 in which all global constraints are of the form $SUM() \leq c$, with objective MAXIMIZE $SUM()$; (3) they may return infeasible packages; (4) they require pre-computation of packages, which are then retrieved at query time using a multi-dimensional index. Package queries also encompass *submodular* optimization queries, whose recent approximate solutions use greedy distributed algorithms [18].

7. CONCLUSIONS

In this paper, 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.

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. Bisschop. *AIMMS Optimization Modeling*. Paragon Decision Technology, 2006.

[4] M. Brucato, J. F. Beltran, A. Abouzied, and A. Meliou. Scalable package queries in relational database systems. *PVLDB*, 9(7):576–587, 2016.

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

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

[7] 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.

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

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

[10] 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.

[11] 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.

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

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

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

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

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

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

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

[19] 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.

[20] 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.

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

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

[23] X. Wang, X. L. Dong, and A. Meliou. Data X-Ray: A diagnostic tool for data errors. In *SIGMOD*, pages 1231–1245, 2015.

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