Noname manuscript No.
(will be inserted by the editor)

Package queries: Efficient and scalable computation of high-order constraints

Matteo Brucato[1] · **Azza Abouzied**[2] · **Alexandra Meliou**[1]

Received: date / Accepted: date

Abstract Traditional database queries follow a simple model: they define constraints that each tuple in the result must satisfy. This model is computationally efficient, as the database system can evaluate the query conditions on each tuple individually. However, many practical, real-world problems require a collection of result tuples to satisfy constraints collectively, rather than individually. In this paper, we present *package queries*, a new query model that extends traditional database queries to handle complex constraints and preferences over answer sets. We develop a full-fledged package query system, implemented on top of a traditional database engine. Our work makes several contributions. (1) We design PaQL, a SQL-based query language that supports the declarative specification of package queries. We prove that PaQL is at least as expressive as integer linear programming, and therefore, evaluation of package queries is NP-hard. (2) We present a fundamental evaluation strategy that combines the capabilities of databases and constraint optimization solvers to derive solutions to package queries. The core of our approach is a set of translation rules that transform a package query to an integer linear program. (3) We introduce an offline data partitioning strategy allowing query evaluation to scale to large data sizes. (4) We introduce SKETCHREFINE, a scalable algorithm for package evaluation, with strong approximation guarantees ($(1 \pm \varepsilon)$-factor approximation). (5) We present a method for parallelizing the REFINE phase of SKETCHREFINE. (6) We present an empirical study of the efficiency gains of providing integer solvers with starting solutions. (7) We present extensive experiments over real-world and benchmark data. The results demonstrate that our methods are effective at deriving high-quality package results, and achieve runtime performance that is an order of magnitude faster than directly using ILP solvers over large datasets.

Keywords Package queries · Integer linear programming · Approximation algorithm · SketchRefine · PaQL

M. Brucato
matteo@cs.umass.edu

A. Abouzied
azza@nyu.edu

A. Meliou
ameli@cs.umass.edu

[1] College of Information and Computer Sciences, University of Massachusetts, Amherst, MA, USA
[2] Computer Science, New York University, Abu Dhabi, UAE

1 Introduction

Traditional, non-recursive database queries rely on a simple evaluation model: they define constraints, in the form of selection predicates, that each tuple in the result must satisfy. This model is computationally efficient,[1] 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 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 [21].

Example 3 (Investment portfolio) A broker wants to identify a set of investment assets to form an investment portfolio. The portfolio should diversify the asset types (e.g., 75% stocks, 25% bonds), limit exposure in certain categories (e.g., 20% energy, 50% technology), and minimize volatility.

[1] The evaluation of non-recursive SQL queries is polynomial with respect to data complexity. When we discuss complexity in this paper, we refer to data complexity.

In these examples, there are some conditions that can be verified on individual data items (e.g., gluten content in a meal), while others need to be evaluated on a collection of items (e.g., total calories). Similar scenarios arise in a variety of application domains, such as product bundles, course selection [31], team formation [2,26], vacation and travel planning [9], and computational creativity [33]. 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 [2,9,26,31].

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

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

Our work addresses three important challenges:

Declarative specification of packages. The first challenge is to support *declarative* specification of packages. SQL enables the declarative specification of properties that result tuples should independently satisfy. In Example 1, it is easy to specify the exclusion of meals with gluten using a regular SQL selection predicate. 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 (Section 2). Our goal is to maintain the declarative power of SQL, while extending its expressiveness to allow for the easy specification of packages.

Evaluation of package queries. The second challenge pertains 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 integer linear programs (Section 2.4). 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 (described in detail in Section 2) as opposed to an integer linear program (Section 3). As the cardinality of the package increases, so does the number of joins, and the runtime of the SQL solution quickly becomes prohibitive: In a small set of 100 tuples from the Sloan Digital Sky Survey dataset [34], 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.

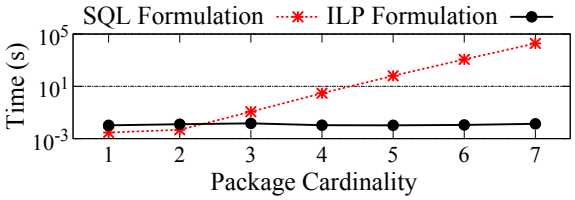

Fig. 1: Traditional database technology is ineffective at package evaluation, and the runtime of the SQL formulation of a package query (Section 2) grows exponentially. In contrast, ILP solvers (Section 3) are more effective.

Performance and scaling to large datasets. The third challenge relates 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 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. Specifically, we make the following contributions:

- We present PaQL (Package Query Language), a declarative language that provides simple extensions to standard SQL to support constraints at the package level. We prove that PaQL is at least as expressive as integer linear programming, which implies that evaluation of package queries is NP-hard (Section 2).
- We present a fundamental evaluation strategy, DIRECT, that combines the capabilities of databases and constraint optimization solvers to derive solutions to package queries. The core of our approach is a set of translation rules that transform a package query to an integer linear program. This translation allows for the use of highly-optimized tools for the evaluation of package queries (Section 3).
- We introduce an offline data partitioning strategy that allows package query evaluation to scale to large data sizes.

The core of our evaluation strategy, SKETCHREFINE, consists of 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 sketched package by solving an integer program within each partition. SKETCHREFINE guarantees a $(1 \pm \varepsilon)$-factor approximation compared to DIRECT, where ε is a flexible parameter of the offline partitioning. (Section 4).

– We present an extensive experimental evaluation on both real-world data and the TPC-H benchmark (Section 6) that shows that our query evaluation method SKETCHREFINE: (1) is able to produce packages an order of magnitude faster than the integer solver used directly on the entire problem; (2) scales up to sizes that the solver cannot manage directly; (3) produces packages of very good quality in terms of objective value; (4) is robust to partitioning built in anticipation of different workloads.
– We design a parallel version of SKETCHREFINE that can efficiently solve queries that require most of the partitions to be accessed. We experimentally show that this type of queries is a worst case for the offline data partitioning used by SKETCHREFINE, and severely impacts the sequential performance of the algorithm.
– We present an empirical study on *preconditioning* solvers with starting solutions. Our results show that seeding solvers with feasible packages can significantly improve the performance of the solver especially on harder queries.

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. In this section, we first describe two ways of expressing package queries in SQL and explain their drawbacks. We then describe PaQL, a declarative query language for specifying packages, and analyze its complexity and expressivity.

2.1 Expressing package queries with SQL

Specifying packages with self-joins. In the limited case of packages with strict cardinality, i.e., 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.sat_fat + R2.sat_fat + R3.sat_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.

2.2 Relation and package semantics

We first introduce some basic notation and describe the semantics of packages. Let U be the *universe* of possible *tuples* of a relation R and $\mathbb{N}$ the set of all the natural numbers, $\mathbb{N} = \{0, 1, \dots\}$. R is a multiset over universe U, denoted as (U, m_{R}), where $m_{\mathsf{R}} : U \to \mathbb{N}$ is a multiplicity function, indicating the number of occurrences of each element of U in R. Throughout the paper, we use the following multiset operators: Given relations R_1 and R_2, $\mathsf{R}_1 \subseteq \mathsf{R}_2$, iff $\forall t \in U : m_{\mathsf{R}_1}(t) \leq m_{\mathsf{R}_2}(t)$; $\mathsf{R}_1 \cup \mathsf{R}_2$ has multiplicity $m_{\mathsf{R}_1 \cup \mathsf{R}_2}(t) = m_{\mathsf{R}_1}(t) + m_{\mathsf{R}_2}(t)$, $\forall t \in U$; $\mathsf{R}_1 \setminus \mathsf{R}_2$ has multiplicity $m_{\mathsf{R}_1 \setminus \mathsf{R}_2}(t) = \max\{0, m_{\mathsf{R}_1}(t) - m_{\mathsf{R}_2}(t)\}$, $\forall t \in U$.

A *package* P, defined over R, is a multiset with multiplicity $m_{\mathsf{P}} : U \to \mathbb{N}$, such that $\forall t \in U : m_{\mathsf{R}}(t) = 0 \implies m_{\mathsf{P}}(t) = 0$.

2.3 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.

PaQL syntax

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

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)
```

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

aggregate functions or SQL subqueries with aggregate functions; both **st_expression** and **obj_expression** can 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(*)
    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. Each tuple, whether or not unique in the input relation, has unbounded multiplicity in the package. 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 the relation Recipes. 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.[2]

The language also permits multiple relations in the FROM clause; in that case, the packages produced will follow the schema of the join result. In the remainder of this paper, we focus on package queries without joins. This is for two reasons: (1) The join operation is part of traditional SQL and can occur before package-specific computations. (2) There are important implications in the consideration of joins that extend beyond the scope of our work. Specifically, materializing the join result is not always necessary, but rather, there are space-time trade-offs and system-level solutions that can improve query performance in the presence of joins. These extensions are orthogonal to the techniques we present in this work.

Although semantically valid, a query like $\mathcal{Q}_2$ would not occur in practice, as most application scenarios expect few, or even exactly one result. We proceed to describe the additional constraints in the example query $\mathcal{Q}$ (Figure 2) that restrict the number of package results.

[2] This syntax slightly differs from the one presented in [6].

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 a tuple has duplicates in the input table (multiplicity greater than 1), repetition restrictions are applied on each individual duplicate. Formally, for input table R, REPEAT ρ, $\rho \geq 0$, implies $\forall t \in U : m_{\mathsf{P}}(t) \leq m_{\mathsf{R}}(t)(1+\rho)$. 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. Tuple repetitions naturally appear in many problems (e.g., Example 3, where multiple copies of the same investment asset can be included in a portfolio). While the PaQL specification allows for an arbitrarily large number of repetitions, we expect that systems will impose a default bound in practice. In this paper, we focus on queries with explicit repetition constraints.

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, e.g., 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, e.g., 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 [7], we described preliminary techniques for returning multiple packages in the absence of optimization objectives, but a thorough study of such methods is left to future work.

While PaQL allows arbitrary aggregate functions in the global predicates and the objective clause, in this work, we only support package queries with *linear* aggregates over numerical variables. A linear aggregate can be a constant or an attribute value multiplied by a constant, or any linear combination thereof. We defer the study of non-linear aggregates and UDFs to future work.

2.4 Expressiveness and complexity of PaQL

Package queries can model a great variety of problems. They are at least as expressive as integer linear programs (ILP), and, therefore, at least as hard.

Theorem 1 (Expressiveness of PaQL) *Every integer linear program can be expressed as a package query in PaQL.*

Proof We prove the result through a reduction from an ILP problem to a PaQL query. The reduction involves two mappings: (1) a mapping from a general ILP instance $\mathcal{I}$ to a PaQL query $\mathcal{Q}_\mathcal{I}$; (2) a mapping from a solution to the ILP problem to a package p. The mappings are such that the solution to the ILP is an optimal solution to $\mathcal{I}$ *iff* p is an optimal package for $\mathcal{Q}_\mathcal{I}$. Let $\mathcal{I}$ be an ILP problem involving n integer variables[3], k linear constraints, and real coefficients a_i, b_{ij} and c_j:

$$\begin{aligned}\mathcal{I}: \max\ & \textstyle\sum_{i=1}^{n} a_i x_i \\ \text{s.t}\ & \textstyle\sum_{i=1}^{n} b_{ij} x_i \le c_j \ \ \forall j = 1,\ldots,k \\ & x_i \ge 0, x_i \in \mathbb{Z} \quad \forall i = 1,\ldots,n\end{aligned}$$

The PaQL query $\mathcal{Q}_\mathcal{I}$ constructed from $\mathcal{I}$ is:

```
Q_I: SELECT      PACKAGE(*) AS P    FROM (
       SELECT a_1 AS attr_obj, b_11 AS attr_1, ..., b_1k AS attr_k
       UNION ···
       SELECT a_n AS attr_obj, b_n1 AS attr_1, ..., b_nk AS attr_k)
     SUCH THAT SUM(P.attr_1) ≤ c_1 AND ...SUM(P.attr_k) ≤ c_k
     MAXIMIZE  SUM(P.attr_obj)
```

Let $\hat{x}$ be an *assignment* to the variables in $\mathcal{I}$. Package p is constructed from $\hat{x}$ by including tuple t_i exactly $\hat{x}_i$ times.

[3] For ease of presentation, we show an ILP with nonnegative variables, but the mapping generalizes to arbitrary integer variables: negative variables negate the corresponding values in the query; for arbitrary bounds on each variable, add cardinality constraints to individual tuples.

($\Rightarrow$) Suppose $\hat{x}$ is an optimal feasible solution to $\mathcal{I}$. Then $\forall j = 1,\ldots,k, \sum_{i=1}^{n} b_{ij}\hat{x}_i \le c_j$ and $\sum_{i=1}^{n} a_i\hat{x}_i$ is maximal. Thus, by construction of p, $\forall j = 1,\ldots,k$, $\mathsf{SUM}(p.\mathsf{attr}_j) = \sum_{i=1}^{n} b_{ij}\hat{x}_i \le c_j$, and $\mathsf{SUM}(p.\mathsf{attr}_{obj}) = \sum_{i=1}^{n} a_i\hat{x}_i$ is maximal. Therefore, p is an optimal package for query $\mathcal{Q}_\mathcal{I}$.
($\Leftarrow$) If p is an optimal package for $\mathcal{Q}_\mathcal{I}$, then, by definition, $\forall j = 1,\ldots,k, \sum_{i=1}^{n} b_{ij}\hat{x}_i \le c_j$ and $\sum_{i=1}^{n} a_i\hat{x}_i$ is maximal. □

As a direct consequence of Theorem 1, we obtain the following result on the complexity of package query evaluation.

Corollary 1 (Complexity of Package Queries) *Package queries are NP-hard.*

In Section 3, we extend the result of Theorem 1 to also show that every PaQL query over any database instance can be encoded as an integer linear program, through a set of translation rules. This transformation is the first step in package evaluation, but, due to the limitations of ILP solvers, it is not efficient or scalable in practice. To make package evaluation practical, we develop SKETCHREFINE (Section 4), a technique that augments the ILP transformation with a partitioning mechanism, allowing package evaluation to scale to large datasets. In Section 7, we show how to parallelize SKETCHREFINE, in order to efficiently answer queries that require most of the partitions to be accessed. Finally, in Section 8, we show how starting solutions can improve the performance of the ILP solver.

3 ILP formulation of package queries

In this section, we present an ILP formulation for package queries. This formulation is at the core of our evaluation methods DIRECT and SKETCHREFINE. The results presented in this section are inspired by the translation rules employed by Tiresias [27] to answer *how-to queries*. However, there are several important differences between how-to and package queries, which we extensively discuss in the overview of the related work (Section 9).

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), $\odot \in \{\le, \ge\}$ a constraint inequality, and $v \in \mathbb{R}$ a constant. For each tuple t_i from R, $1 \le i \le n$, the ILP problem includes a nonnegative integer variable x_i, $x_i \ge 0$, 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 $0 \le x_i \le \rho + 1$.

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	$x_5 = 1$

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

Fig. 3: Example ILP formulation and solution for query $\mathcal{Q}$, 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.**

Base predicate: Let β be a base predicate, e.g., R.gluten = 'free', and R_β the relation containing tuples from R satisfying β. We encode β by setting $x_i = 0$ for every tuple $t_i \notin \mathsf{R}_\beta$.

Global predicate: Each global predicate in the SUCH THAT clause takes the form $f(\mathsf{P}) \odot v$. For each such predicate, we derive a linear function $f'(\bar{x})$ over the integer variables. A cardinality constraint $f(\mathsf{P}) = \mathsf{COUNT}(\mathsf{P}.*)$ is translated into a linear function $f'(\bar{x}) = \sum_i x_i$. A summation constraint $f(\mathsf{P}) = \mathsf{SUM}(\mathsf{P}.\mathsf{attr})$ is translated into a linear function $f'(\bar{x}) = \sum_i (t_i.\mathsf{attr}) x_i$. We further illustrate the translation with two non-trivial examples:

- $\mathsf{AVG}(\mathsf{P}.\mathsf{attr}) \leq v$ is translated as
$$\textstyle\sum_i (t_i.\mathsf{attr}) x_i / \sum_i x_i \leq v \;\equiv\; \sum_i (t_i.\mathsf{attr} - v) x_i \leq 0$$
- (SELECT COUNT$(*)$ FROM P WHERE P.carbs > 0) $\geq$ (SELECT COUNT$(*)$ FROM P WHERE P.protein ≤ 5) is translated as
$$\textstyle\sum_i (\mathbb{1}_{\mathsf{R_c}}(t_i) - \mathbb{1}_{\mathsf{R_p}}(t_i)) x_i \geq 0$$
where
$\mathsf{R_c} := \{t_i \in \mathsf{R} \mid t_i.\mathsf{carbs} > 0\}$
$\mathsf{R_p} := \{t_i \in \mathsf{R} \mid t_i.\mathsf{protein} \leq 5\}$
$\mathbb{1}_{\mathsf{R_c}}(t_i) := 1$ if $t_i \in \mathsf{R_c}$; 0 otherwise
$\mathbb{1}_{\mathsf{R_p}}(t_i) := 1$ if $t_i \in \mathsf{R_p}$; 0 otherwise.

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

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

We call the relations R_β, R_c, and R_p described above *base relations*. This formulation, together with Theorem 1, shows that package queries with linear constraints and linear objective functions correspond *exactly* to ILP problems.

Example 4 (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. This can significantly reduce the size of the problem.
3. **ILP execution.** We employ an off-the-shelf ILP solver, as a black box, to get a solution 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 5 (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 in the output package. Thus, the result of $\mathcal{Q}$ is the package: $\{t_2, t_3, t_5\}$.

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, integer linear programming is a notoriously hard problem, and modern ILP solvers use algorithms, such as *branch-and-cut* [30], 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 due to solvers using too many resources (main memory, virtual memory, CPU time), eventually thrashing the entire system.

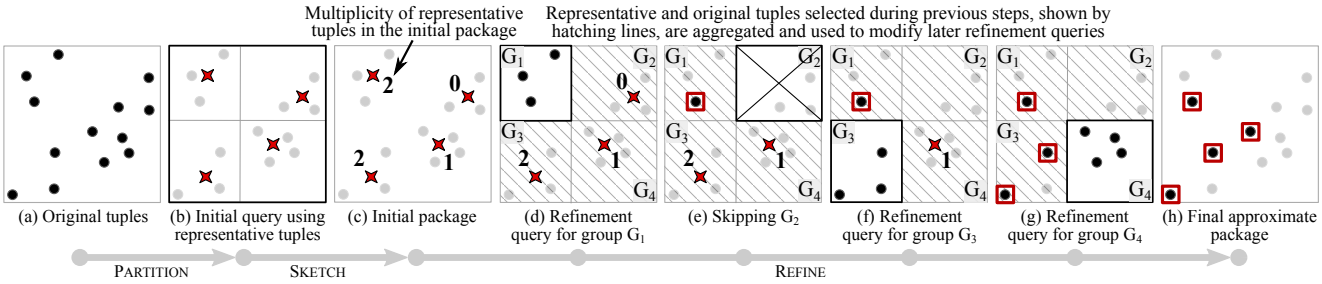

Fig. 4: The original tuples (a) are partitioned into four groups and a representative is constructed for each group (b). The initial sketch package (c) contains only representative tuples, with possible repetitions up the size of each group. The refine query for group G_1 (d) involves the original tuples 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 are used while refining the solution for the remaining groups (f and g). The final approximate package (h) contains only original tuples.

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 the DIRECT approach. Our implementation of SKETCHREFINE uses an ILP solver as its underlining black box for solving the smaller queries; however, SKETCHREFINE is more general in that it can be used to scale *any other black-box solution for solving package queries.* Further, we prove that SKETCHREFINE is guaranteed to always produce feasible packages with an approximate objective value (Section 5.1).

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. This partitioning is performed offline (not at query time), and our experiments show that SKETCHREFINE remains very effective even with partitionings that do not match the query workload (Section 6.2.3). In our implementation, we partition data using k-dimensional quad trees [13], but other partitioning schemes are possible.

Algorithm 1 Scalable package query evaluation

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

2. **Sketch (Section 4.2.1).** SKETCHREFINE sketches an initial package by evaluating the package query only over the set of representative tuples.
3. **Refine (Section 4.2.2).** Finally, SKETCHREFINE transforms the initial package into a complete package by replacing each representative tuple with some of the original tuples from the same group, one group at a time.

SKETCHREFINE always constructs *approximate feasible* packages, i.e., packages that satisfy all the query constraints, but with a possibly sub-optimal objective value that is guaranteed to be within certain approximation bounds (Section 5.1). SKETCHREFINE may suffer from *false infeasibility*, which happens when the algorithm reports a feasible query to be infeasible. The probability of false infeasibility is, however, low and bounded (Section 5.2).

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 $G_i \subseteq \mathsf{R}$, $1 \leq i \leq m$, has a representative tuple $\tilde{t}_i$, which may not always appear in R. We denote the partitioned space with $\mathcal{P} = \{(G_i, \tilde{t}_i) \mid 1 \leq i \leq m\}$. 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 partitioning attributes can be any subset of the numerical attributes of R.

Definition 1 (Size threshold, τ) The *size threshold* τ, $1 \leq \tau \leq n$, restricts the size of each partitioning group G_i, $1 \leq i \leq m$, to a maximum of τ original tuples, i.e., $|G_i| \leq \tau$.

Definition 2 (Diameter bounds) The *diameter* $d_{ij} \geq 0$ of a group G_i, $1 \leq i \leq m$, on attribute $\mathtt{attr}_j$, $1 \leq j \leq k$, is the greatest absolute distance between all pairs of tuples within group G_i:

$$d_{ij} = \max_{t_1,t_2 \in G_i} |t_1.\mathtt{attr}_j - t_2.\mathtt{attr}_j| \quad (1)$$

The *diameter bounds* $\omega_{ij} \geq 0$, $1 \leq i \leq m$, $1 \leq j \leq k$, require all diameters to be bounded by $d_{ij} \leq \omega_{ij}$.

The size threshold, τ, affects the number of partitions, m: a lower τ leads to smaller partitions, but more of them (larger m). As we discuss later (Section 4.2), for best response time of SketchRefine, τ should be set so that both m and τ are small. Our experiments show that a proper setting can lead to an order of magnitude improvement in query response time (Section 6.2.2).

The diameter bounds, ω_{ij}, are not required, but they can be enforced to ensure a desired approximation guarantee (Section 5.1). 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. 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 (Section 6.2.4), and it is a very common characteristic of most approximation schemes [36].

Partitioning method

Different methods can be used for partitioning. Our implementation is based on k-dimensional *quad-tree indexing* [13]. 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 $t.\mathtt{gid} = i$ iff 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 $\mathtt{gid} = 1$ for all tuples. Then, it recursively proceeds as follows:

- The procedure computes the sizes and diameters of the current groups via a query that groups tuples by their gid value. The same group-by query also computes the *centroid* tuple of each group. The centroid is computed by averaging the tuples in the group on each of the partitioning attributes.
- If group G_i has more tuples than the size threshold, or a diameter larger than the allowed bound, the tuples in group G_i are partitioned into 2^k subgroups, where k is the number of partitioning attributes. The group's centroid is the split point to generate sub-partitions: tuples that reside in the same sub-partition are grouped together.

Our method recursively executes two SQL queries on each subgroup that violates the size or the diameter conditions.

Stored representatives

After partitioning, a group-by query computes the minimum, maximum, and average values of all the partitioning attributes, and stores them in a relational table. At query time, the algorithm loads representatives from this table, selecting only one aggregate type per query attribute (either minimum, maximum of average), into a *representative relation* $\tilde{\mathsf{R}}(\mathsf{gid}, \mathtt{attr}_1, \ldots, \mathtt{attr}_k)$. To ensure approximation guarantees (Section 5.1), the maximum (minimum, resp.) value is chosen for a maximization (minimization, resp.) query. For all other attributes, the algorithm picks the average value.

Alternative partitioning approaches

We experimented with different clustering algorithms, such as *k-means* [18], *hierarchical clustering* [24] and DBSCAN [11], using off-the-shelf libraries such as *Scikit-learn* [32]. Existing clustering algorithms present various problems: First, they tend to vary substantially in the properties of the generated clusters. In particular, none of the existing clustering techniques can natively generate clusters that satisfy the size threshold τ and diameter limits ω_{ij}. In fact, most of the clustering algorithms take as input the *number of clusters* to generate, without offering any means to restrict the size of each cluster nor their diameter. Second, existing implementations only support in-memory cluster computation, and DBMS-oriented implementations usually need complex and inefficient queries. On the other hand, space partitioning techniques from multi-dimensional indexing, such as *k-d trees* [3] and *quad trees* [13], can be more easily adapted to satisfy the size and diameter limits, and to work within the database: our partitioning method works directly on the input table via simple SQL queries.

Finally, partitioning could be dynamically generated at query time: By maintaining the entire hierarchical structure of the quad-tree index, one can traverse the index at query time to generate the coarsest partitioning that satisfies the required size and diameter limits. However, index traversal incurs additional overhead at query time, compared to using a precomputed static partitioning.

One-time cost
Partitioning is an expensive procedure. To avoid paying its cost at query time, the dataset is partitioned in advance and used to answer a workload of package queries. For a known workload, our experiments show that partitioning the dataset on the union of all query attributes provides the best performance in terms of query evaluation time and approximation error for the computed answer package (Section 6.2.3). We also demonstrate that our query evaluation approach is robust to a wide range of partition sizes, and to imperfect partitions that cover more or fewer attributes than those used in a particular query. This means that, even without a known workload, a partitioning performed on all of the data attributes still provides good performance.

Enforcing a diameter limit guarantees the theoretical approximation bounds of SKETCHREFINE (Section 5.1). However, partitioning only with a size threshold can also achieve good quality in practice: Since partitioning splits a group on its centroid, the resulting sub-partitions will naturally have smaller diameters. Our experiments (Section 6) show that partitioning on a size threshold alone results in good approximations while reducing the offline partitioning cost: Meeting a size threshold requires fewer partitioning iterations than meeting a diameter limit especially if the dataset is sparse across the attribute domains.

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.

4.2.1 SKETCH

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

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

The new global constraints, highlighted in bold, ensure that every representative tuple does not appear in $p_{\mathcal{S}}$ more times than the size of its group, G_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 simply omitted from $\mathcal{Q}(\tilde{\mathsf{R}})$ if the original query does not contain a repetition constraint.

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

The SKETCH procedure *fails* if the sketch query $\mathcal{Q}(\tilde{\mathsf{R}})$ is infeasible, in which case SKETCHREFINE reports the original query $\mathcal{Q}$ as infeasible (Algorithm 1). This may constitute *false infeasibility*, if $\mathcal{Q}$ is actually feasible. In Section 5.2, we show that the probability of false infeasibility is low and bounded, and we present simple methods to avoid this outcome.

4.2.2 REFINE

Using the sketched solution over the representative tuples, the REFINE procedure iteratively replaces the representative tuples with tuples from the original relation R, until no more representatives are present in the package. The algorithm *refines* the sketch package $p_{\mathcal{S}}$ one group at a time. For a group G_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:

$\mathcal{Q}_i(p_{\mathcal{S}})$: SELECT PACKAGE$(*)$ AS p_i
FROM G_i REPEAT 0
WHERE G_i.gluten = 'free'
SUCH THAT
COUNT$(p_i.*)$ + **COUNT**$(\bar{p}_i.*) = 3$ AND
SUM$(p_i$.kcal) + **SUM**$(\bar{p}_i$.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

directly, and thus we can solve this package query using the DIRECT method (Section 3.2). Similarly to the sketch query, if τ is too large, we can solve this query recursively with SKETCHREFINE: the tuples in group G_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

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 (Section 5.1).

Algorithm 2 details the REFINE procedure. The algorithm logically traverses a *search tree* (which is never constructed, but is the result of recursive calls and bactracking), where each node corresponds to a unique sketch package $p_\mathcal{S}$. The traversal starts from the *root*, corresponding to the initial sketch package, where no groups have been refined ($\mathcal{S} = \mathcal{P}$), and finishes at the first encountered *leaf*, corresponding to a complete package ($\mathcal{S} = \emptyset$). The algorithm terminates as soon as it encounters a complete package, which it returns (line 4). The algorithm maintains a set of failed groups, $\mathcal{F}$, initially empty (line 2), and assumes a (initially random) refinement order for all groups in $\mathcal{S}$, stored in a priority queue $\mathcal{U}$ (line 6). It then tries to solve the refine query corresponding to each of the groups in the queue (line 12). When a refine query succeeds, the algorithm recursively proceeds with the next group in the queue (lines 13-18). If any of the refine queries fails, the failing group is added to $\mathcal{F}$, and the algorithm immediately backtracks, reporting the failure to the parent node in the search tree (lines 25-29). Failures can occur at any depth of the traversal. If a recursive call fails, all the failing groups ($\mathcal{F}'$) are prioritized (lines 19-22).

Theorem 2 (Correctness of REFINE**)** *A package produced by* REFINE *is guaranteed to satisfy the query constraints.*

The theorem follows from the fact that, by construction, the refine query, $\mathcal{Q}_i(p_\mathcal{S})$, identifies tuples replacements for the representatives that do not break the overall constraints of the original query.

Algorithm 2 Greedy backtracking REFINE

input:
- $\mathcal{Q}$: the package query to be evaluated
- $\mathcal{P} = \{(G_1, \tilde{t}_1), \ldots, (G_m, \tilde{t}_m)\}$: partitioning groups
- $\mathcal{S}$: partitioning groups yet to be refined (initially $\mathcal{S} = \mathcal{P}$)
- $p_\mathcal{S}$: the refining package (initially the result of SKETCH)

output: a feasible package containing only tuples, or failure

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

Let $T(\tau)$ be the time taken by the black box (in our case, DIRECT using an ILP solver) to solve a problem of size τ. We express the time complexity of the refine procedure as a function of $T(\tau)$ and m, the number of partitions used by SKETCHREFINE. 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 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 a factorial number of calls to the solver, $O(T(\tau)m!)$. Our experiments show that the best case is the most common and backtracking occurs infrequently.

False infeasibility and hybrid sketch queries

For a feasible query $\mathcal{Q}$, false negatives, or *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 Theorem 4 establishes in Section 5.2.

In our evaluation, we use a small heuristic modification to SKETCHREFINE to deal with these cases, which creates a hybrid query by merging the sketch query $\mathcal{Q}(\tilde{\mathsf{R}})$ with one of the refine queries. The *hybrid* sketch query, executed in place of the original sketch query, selects tuples from a group and, at the same time, representative tuples from all the remaining groups. This simple technique can greatly reduce false infeasibility by circumventing three potential cases of failure: (1) The original sketch query, $\mathcal{Q}(\tilde{\mathsf{R}})$, may be infeasible due to a bad representative from one of the groups. An hybrid sketch query over that group could render the sketch phase possible. (2) If a group fails in a later refine stage, solving that group upfront with a hybrid sketch query could render the group's problem feasible, thanks to having representatives for the other groups. (3) If a group fails in a later refine stage, a hybrid sketch query on a different group could avoid selecting representatives for the failing group altogether. The algorithm tries a hybrid sketch query on each group whenever the original sketch query is infeasible or when all refines fail; it then proceeds normally if one of the hybrid queries is feasible. Hybrid sketch proves extremely effective on our experimental workload (Section 6): SKETCHREFINE with hybrid sketch does not encounter even a single case of false infeasibility, i.e., there is no query for which DIRECT produces a solution but SKETCHREFINE does not.

5 Theoretical analysis of SKETCHREFINE

SKETCHREFINE scales package evaluation by breaking the problem into smaller, manageable subproblems: the SKETCH phase evaluates a package query over the representative tuples of the partitions, and the REFINE phase evaluates package queries over each partition. This scalability comes at the price of accuracy. A package returned by SKETCHREFINE is guaranteed to satisfy all the query constraints, but it may have a worse objective value than the package produced by DIRECT evaluation. Moreover, SKETCHREFINE may incorrectly determine that a package query is infeasible, when in fact it has a solution (false infeasibility). In this section, we provide a theoretical analysis of the quality of results produced by SKETCHREFINE. Specifically, we present two theoretical results. First, we show that SKETCHREFINE offers strong approximation guarantees: a package produced by SKETCHREFINE is guaranteed to be within a $(1\pm\varepsilon)$-factor from the package produced by DIRECT. Second, we show that SKETCHREFINE fails to produce a package to a feasible query (false infeasibility) with low probability.

5.1 Approximation guarantees

DIRECT and SKETCHREFINE employ a black-box solver to evaluate either the original query (DIRECT), or the subqueries (the sketch and refine queries of SKETCHREFINE). If the solver is exact, then DIRECT returns optimal solutions, and the approximation guarantees of SKETCHREFINE are with respect to the true optimal. In general however, solvers may not be exact (e.g., ILP solvers typically provide approximations), in which case the approximation bound of SKETCHREFINE is with respect to the approximation of the solver. SKETCHREFINE allows control of its approximation bounds through its offline partitioning. Specifically, we prove that, for a desired approximation parameter ε, we can derive diameter bounds ω_{ij} (for each partitioning group G_i and attribute attr_j) for the offline partitioning that guarantee that the solution produced by SKETCHREFINE (if any) has objective value $(1\pm\varepsilon)$-factor close to the objective value of the solution produced by the solver for the same query.

Theorem 3 (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:*

$$\omega_{ij}=\min_{t\in G_i}\{|1-\beta|\cdot|t.\mathsf{attr}_j|\},\quad \forall i\in[1,m],\ \forall j\in[1,k] \qquad (2)$$

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

We present the proof of the theorem for the case of maximization queries. The minimization case follows analogous reasoning. Without loss of generality, we consider a feasible package query $\mathcal{Q}$ with a summation constraint on each of the k attributes, $\mathsf{SUM}(\mathsf{attr}_j)\leq U_j$, $j\in[1,k]$, and a maximization objective on $\mathsf{SUM}(\mathsf{attr}_{obj})$. A COUNT constraint is a special case of a SUM over an attribute that is equal to 1. Partitioning over this attribute would result in groups with zero diameter (the value of the attribute for all tuples in the group is the same). Therefore, with respect to this attribute, representatives are exact. Essentially, COUNT constraints do not affect the approximation of the result.

We prove Theorem 3 in two steps. First, we show that the initial SKETCH package approximates the optimal package by a factor β. Second, we show that the final package returned by the REFINE procedure approximates the initial SKETCH package by a factor β as well. Thus, the final result of SKETCHREFINE approximates the optimal package by a factor of β^2. We conclude the proof by showing an explicit value for β as a function of ε. The proof requires two lemmas (Lemma 2 and Lemma 3 below). The first lemma shows that if a package satisfies $\mathcal{Q}$, replacing the tuples in the package with their representative tuples generates a package

that satisfies a relaxed version of $\mathcal{Q}$, where each constraint is relaxed by a factor β. Below, we define such relaxed queries as β-*relaxations*. The second lemma shows that if a package p_1 optimizes $\mathcal{Q}$ and another package p_2 optimizes its β-relaxation, then the objective value of p_1 cannot be worse than the objective value of p_2 by more than a factor β.

We first introduce some needed notation and definitions. Given a package p, we denote the summation of its tuples on attribute attr with $\mathsf{SUM}(p.\mathsf{attr})$, and its objective value with $\mathsf{OBJ}(p)$, where $\mathsf{OBJ}(p) = \mathsf{SUM}(p.\mathsf{attr}_{obj})$. We now proceed to define the concepts of *ordering*, and *feasible*, *optimal*, and *approximate* packages, that are at the core of the proof.

Definition 3 (Package ordering $\succeq$) A package p_1 *dominates* a package p_2, denoted by $p_1 \succeq p_2$, iff the objective value of p_1 is at least as good the objective value of p_2: $\mathsf{OBJ}(p_1) \geq \mathsf{OBJ}(p_2)$. With slight abuse of notation, we write $p_1 \succeq \beta\ p_2$ to denote that the objective value of p_1 is as least as good as the objective value of p_2 by a factor β.

Definition 4 (Feasible package $\models$) We say that a package p is *feasible* for $\mathcal{Q}$, denoted by $p \models \mathcal{Q}$, iff for all $1 \leq j \leq k$: $\mathsf{SUM}(p.\mathsf{attr}_j) \leq U_j$.

Definition 5 (Optimal package $\models^*$) A package p is *optimal* for $\mathcal{Q}$, denoted by $p \models^* \mathcal{Q}$, iff $p \models \mathcal{Q}$ and for all $p' \models \mathcal{Q}$, $p \succeq p'$.

Definition 6 (β-approximation) A package p is a β-*approximation* for query $\mathcal{Q}$ if $p \models \mathcal{Q}$ and for all $p' \models \mathcal{Q}$, $p \succeq \beta\ p'$.

Definition 7 (β-relaxation) The β-*relaxation* of query $\mathcal{Q}$, denoted by $\mathcal{Q}_\beta$, is a query with the same objective function as $\mathcal{Q}$, and with k global constraints, for all $1 \leq j \leq k$:

$$\mathsf{SUM}(\mathsf{attr}_j) \leq \beta^{-1} U_j$$

Definition 8 (Representative projection π) The representative projection of a package p, denoted by $\pi(p)$, is a function that substitutes each tuple in p with its representative tuple.

Because representative tuples have the best value on the objective attribute attr_{obj} of all the tuples in its group, π satisfies the following property:

Property 1 The representative projection of a package dominates the package: $\pi(p) \succeq p$.

Before stating Lemma 2, we introduce another intermediate result. Lemma 1 states that the diameter conditions of Equation (2) guarantee that all the tuples in a group are "close" to each other by a factor no larger than β. We refer to this as β-*closeness*, and we generalize this concept to pairs of packages: two packages are β-close to each other if their sums (on any attribute) are close to each other by a factor β.

Definition 9 (β-closeness) Any two tuples t_1 and t_2 are β-close to each other iff for all $1 \leq j \leq k$:

$$t_1.\mathsf{attr}_j \geq \beta\, t_2.\mathsf{attr}_j \ \text{ and } \ t_2.\mathsf{attr}_j \geq \beta\, t_1.\mathsf{attr}_j$$

Any two packages p_1 and p_2 are β-close to each other iff for all $1 \leq j \leq k$:

$$\mathsf{SUM}(p_1.\mathsf{attr}_j) \geq \beta\, \mathsf{SUM}(p_2.\mathsf{attr}_j) \ \text{ and}$$
$$\mathsf{SUM}(p_2.\mathsf{attr}_j) \geq \beta\, \mathsf{SUM}(p_1.\mathsf{attr}_j)$$

Lemma 1 *If the partitioning satisfies the diameter limits of Equation* (2)*, then all tuples within the same group are* β*-close to each other.*

Proof Consider any group G_i, any attribute attr_j, any pair of tuples t_1, t_2 in G_i. First, $|1-\beta| = (1-\beta)$ as $\beta \in [0,1)$. By Equation (1), $t_1.\mathsf{attr}_j \geq t_2.\mathsf{attr}_j - d_{ij}$. By Equation (2), $d_{ij} \leq (1-\beta)|t_2.\mathsf{attr}_j|$. Thus, either (i) $-d_{ij} \geq (1-\beta)\ t_2.\mathsf{attr}_j$ or (ii) $-d_{ij} \geq (\beta-1)\ t_2.\mathsf{attr}_j$:
If (i): $t_1.\mathsf{attr}_j \geq t_2.\mathsf{attr}_j + (1-\beta)\, t_2.\mathsf{attr}_j > \beta\, t_2.\mathsf{attr}_j$
If (ii): $t_1.\mathsf{attr}_j \geq t_2.\mathsf{attr}_j + (\beta-1)\, t_2.\mathsf{attr}_j = \beta\, t_2.\mathsf{attr}_j$ □

The following lemma states that the representative projection of a feasible package for query $\mathcal{Q}$ satisfies a β-relaxed version of the same query.

Lemma 2 (Representative projection relaxation) *For any package p:* $p \models \mathcal{Q} \implies \pi(p) \models \mathcal{Q}_\beta$.

Proof By hypothesis, for all $1 \leq j \leq k$, $U_j \geq \mathsf{SUM}(p.\mathsf{attr}_j)$. By Lemma 1, $\mathsf{SUM}(p.\mathsf{attr}_j) \geq \beta\, \mathsf{SUM}(\pi(p).\mathsf{attr}_j)$. Therefore, $\mathsf{SUM}(\pi(p).\mathsf{attr}_j) \leq \beta^{-1}\, U_j$. □

Lemma 3 (β-relaxation approximation) *For any packages* p_1*,* p_2*:* $p_1 \models^* \mathcal{Q}$ *and* $p_2 \models^* \mathcal{Q}_\beta \implies p_1 \succeq \beta\ p_2$.

Proof Because $p_2 \models \mathcal{Q}_\beta$, for all $1 \leq j \leq m$, $\mathsf{SUM}(p_2.\mathsf{attr}_j) \leq \beta^{-1} U_j$. Thus, $\beta\ \mathsf{SUM}(p_2.\mathsf{attr}_j) \leq U_j$ and therefore, with abuse of notation, $\beta\ p_2 \models \mathcal{Q}$. Since $p_1 \models^* \mathcal{Q}$, $p_1 \succeq \beta\ p_2$. □

We are now ready to prove Theorem 3.

Proof (of Theorem 3) Let the initial sketch package be denoted by $p^{(0)}$. Suppose, without loss of generality, that the algorithm refines the initial package in the order: $G_1, G_2, \ldots, G_m$. Let $p^{(i)}$ denote the intermediate refined package produced at the i-th iteration of the algorithm. The final complete package returned by the algorithm is thus $p^{(m)}$. Let $p^* \models^* \mathcal{Q}$ be an optimal package. To prove the theorem, we show that there exist a β such that $p^{(m)} \succeq \beta^2\ p^*$. We do so in two steps:

$$p^{(0)} \succeq \beta\ p^* \quad \text{(SKETCH)} \qquad p^{(m)} \succeq \beta\ p^{(0)} \quad \text{(REFINE)}$$

(SKETCH) First, notice that $p^{(0)} \models^* \mathcal{Q}$ because $p^{(0)}$ optimizes the SKETCH query $\mathcal{Q}(\tilde{\mathsf{R}})$ (Section 4.2.1), which has

identical constraints and maximization objective as $\mathcal{Q}$. Consider $p' \models^* \mathcal{Q}_\beta$, the optimal package for the relaxed query $\mathcal{Q}_\beta$ constructed with representative tuples. By Lemma 3, we know that $p^{(0)} \succeq \beta\, p'$. By Lemma 2, we also know that $\pi(p^*) \models \mathcal{Q}_\beta$. Since p' is the optimal package for $\mathcal{Q}_\beta$, $p' \succeq \pi(p^*)$. Finally, by Property 1 of π, we also know that $\pi(p^*) \succeq p^*$. Putting these together, we have that:

$$p^{(0)} \succeq \beta\, p' \succeq \beta\, \pi(p^*) \succeq \beta\, p^*$$

(REFINE) Consider package $p_i^{(i)}$, the solution the i-th REFINE query (Section 4.2.2) computed at the i-th iteration of the algorithm. Clearly, $p_i^{(i)} \models^* \mathcal{Q}_i(p_S)$ because it optimizes the REFINE query. SKETCHREFINE maintains this solution for group G_i until the end of the procedure, thus $p_i^{(m)} = p_i^{(i)}$ and, therefore, $p_i^{(m)} \models^* \mathcal{Q}_i(p_S)$. Consider now $p_i^{(0)}$, the set of representatives computed during the SKETCH phase for group G_i. Because of Lemma 1, during the course of the algorithm, the constraints of a REFINE query can only vary by a factor β. Thus, it must be that $p_i^{(0)} \models \mathcal{Q}_i(p_S)_\beta$. Let $p'' \models^* \mathcal{Q}_i(p_S)_\beta$ be the optimal package for the relaxed version of $\mathcal{Q}_i(p_S)$. Then, $p'' \succeq p_i^{(0)}$. Also, by Lemma 3, we know that $p_i^{(m)} \succeq \beta\, p''$. Putting these together, we have that:

$$p_i^{(m)} \succeq \beta\, p'' \succeq \beta\, p_i^{(0)}$$

Finally, because $p^{(m)} = \sum_{i=1}^m p_i^{(m)}$ and $p^{(0)} = \sum_{i=1}^m p_i^{(0)}$, by linearity of sum we have that $p^{(m)} \succeq \beta\, p^{(0)}$.

Thus, for $\beta = (1-\varepsilon)^{\frac{1}{2}}$ ($\beta = (1+\varepsilon)^{\frac{1}{2}}$, resp.), we get approximation factor $1-\varepsilon$ ($1+\varepsilon$, resp.). □

The theorem implies that, in order to obtain $(1\pm\varepsilon)$-factor approximation, the partitioning must satisfy the following diameter conditions for each group G_i and attribute $\mathtt{attr}_j$:

$$\omega_{ij} = \begin{cases} \min\limits_{t\in G_i} |1-(1-\varepsilon)^{\frac{1}{2}}| \cdot |t.\mathtt{attr}_j| & \text{for maximization} \\ \min\limits_{t\in G_i} |1-(1+\varepsilon)^{\frac{1}{2}}| \cdot |t.\mathtt{attr}_j| & \text{for minimization} \end{cases}$$

5.2 False infeasibility bounds

The following theorem establishes that the probability that SKETCHREFINE will fail to find a solution to a feasible query is low and bounded.

Theorem 4 *For any query $\mathcal{Q}$ and any random package P, if $\mathsf{P} \models \mathcal{Q}$, then with high probability: (1) the* SKETCH *query $\mathcal{Q}(\tilde{\mathsf{R}})$ is feasible; (2) all* REFINE *queries $\mathcal{Q}_i(p_S)$, $1 \le i \le m$, are feasible. Thus,* SKETCHREFINE *returns a feasible result.*

Proof (1) We first show that the sketch query $\mathcal{Q}(\tilde{\mathsf{R}})$ is feasible with high probability.

Suppose, by hypothesis, that $\mathsf{P} \models \mathcal{Q}$. Thus, P satisfies all constraints of $\mathcal{Q}$. Let $\mathsf{SUM}(\mathsf{A})$ be *any* such constraint, where A is either a constant, an attribute from the schema of the input relation R, or a linear combination of attributes of R. Because P is random, its representative projection $\pi(\mathsf{P})$ (Definition 8), constructed from P by replacing tuples with representatives, is also a random package. Thus, both $\mathsf{SUM}(\mathsf{P.A})$ and $\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})$ are random variables. We show that, with high probability, $\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})$ does not differ from the expected $\mathsf{SUM}(\mathsf{P.A})$ and, thus, since P is feasible, so is $\pi(\mathsf{P})$. This implies that the sketch query $\mathcal{Q}(\tilde{\mathsf{R}})$ is feasible with high probability, as at least one solution to it exists, namely $\pi(\mathsf{P})$.

As a first step, we apply Hoeffding's inequality [19] to $\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})$. For all $c > 0$, let $\gamma_{c,\mathsf{P}} = 2\exp\left(-\frac{2c^2}{|\mathsf{P}|(\mathsf{MAX}(\mathsf{A})-\mathsf{MIN}(\mathsf{A}))^2}\right)$. Hoeffding's inequality establishes that the probability of $\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})$ deviating from its expectation by more than c is bounded by a term, $\gamma_{c,\mathsf{P}}$, that is exponentially small in c and $|\mathsf{P}|$:

$$Pr[|\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A}) - E[\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})]| \ge c] \le \gamma_{c,\mathsf{P}} \qquad (3)$$

Let A be the random variable corresponding to the value of attribute A of a random tuple in P, and let $E[A]$ be its expected value. Similarly, let $\tilde{A}$ be the random variable corresponding to a random representative tuple in $\pi(\mathsf{P})$, and $E[\tilde{A}]$ its expected value. Finally, let G be the group a random representative tuple in $\pi(\mathsf{P})$ belongs to. Because representative tuples are the centroids (mean) of all the tuples in their group along the attributes involved in the constraints, we have that:

$$E[\tilde{A}] = E\left[\tfrac{1}{|G|}\textstyle\sum_G A\right] = \tfrac{1}{|G|}\textstyle\sum_G E[A] = E[A] \qquad (4)$$

The expected sum over package $\pi(\mathsf{P})$ is therefore:

$$E[\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A})] = \textstyle\sum_{\mathsf{P}} E[\tilde{A}] = \sum_{\mathsf{P}} E[A] = E[\mathsf{SUM}(\mathsf{P.A})]$$

Thus Equation (3) becomes:

$$Pr[|\mathsf{SUM}(\pi(\mathsf{P}).\mathsf{A}) - E[\mathsf{SUM}(\mathsf{P.A})]| \ge c] \le \gamma_{c,\mathsf{P}} \qquad (5)$$

Equation (5) shows that the probability that the sum of A over $\pi(\mathsf{P})$ differs from the expected sum over P by more than $c > 0$ is bounded. Since $\mathsf{SUM}(\mathsf{P.A})$ is feasible (by hypothesis), so is $\mathsf{SUM}(\pi(\mathsf{P.})\mathsf{A})$, and the sketch query is feasible on this constraint with high probability. This is independently true for all query constraints. Thus, the probability of the overall sketch query being infeasible is one minus the probability of all constraints being feasible. With k constraints, this probability (sketch being infeasible) is small and bounded by $1-(1-\gamma_{c,\mathsf{P}})^k$. This term is exponentially small in c and $|\mathsf{P}|$, so, with high probability, the sketch query $\mathcal{Q}_i(p_S)$ is feasible.

(2) Now, we show that all refine queries are feasible with high probability. Equation 4 allows reasoning about each refine query independently, as replacing representatives with tuples does not change the expected sum in each group.

Let P_i be the tuples in P that belong to group G_i. Then, $\pi(\mathsf{P}_i)$ is the set of representatives in $\pi(\mathsf{P})$ that

belong to group G_i. We apply Hoeffding's inequality on $\mathsf{SUM}(\mathsf{P}_i.\mathsf{A})$, obtaining an equation similar to Equation (3). The proof now follows the same steps as the proof of (1), now applied on $\mathsf{SUM}(\mathsf{P}_i.\mathsf{A})$. From Equation (4), we have that $E[\mathsf{SUM}(\mathsf{P}_i).\mathsf{A})] = E[\mathsf{SUM}(\pi(\mathsf{P}_i))]$. This results in an equation similar to (5), showing that, if $\pi(\mathsf{P}_i)$ is feasible for the *i*-th refine query, then P_i must also be feasible for the same query. When $\pi(\mathsf{P})$ is feasible, $\pi(\mathsf{P}_i)$ is a feasible package for the *i*-th refine query $\mathcal{Q}_i(p_\mathcal{S})$, otherwise the sketch query would be infeasible. This is independently true for all constraints, and the probability of the overall query being infeasible, with *k* constraints, is bounded by $1-(1-\gamma_{c,\mathsf{P}})^k$. Thus, for every group G_i, with high probability, the REFINE query $\mathcal{Q}_i(p_\mathcal{S})$ is feasible. □

Let the *selectivity* of a query be the probability of a random package being *infeasible*. Thus, the lower the selectivity of $\mathcal{Q}$, the higher the probability $Pr[\mathsf{P} \models \mathcal{Q}]$. Therefore, a consequence of Theorem 4 is that the lower the selectivity of $\mathcal{Q}$, the higher the probability that $\mathcal{Q}(\tilde{\mathsf{R}})$ and all $\mathcal{Q}_i(p_\mathcal{S})$ are feasible, which implies that SKETCHREFINE will eventually find a feasible package with high probability as well.

6 Experimental evaluation of SKETCHREFINE

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

6.1 Experimental setup

Software
We implemented our package evaluation system as a layer on top of a traditional relational DBMS. The data itself resides in the database, and the system interacts with the DBMS via SQL when it needs to perform operations on the data. We use PostgreSQL v9.3.9 for our experiments. The core components of our evaluation module are implemented in Python 2.7. The PaQL parser is generated in C++ from a context-free grammar, using GNU Bison [15]. We represent a package in the relational model as a standard relation with schema equivalent to the schema of the input relation. A package is materialized into the DBMS only when necessary (for example, to compute its objective value).

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

Fig. 5: Size of the tables used in the TPC-H benchmark.

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

Fig. 6: Partitioning time for the two datasets, using the workload attributes and with no diameter condition.

We employ IBM's CPLEX [20] v12.6.1 as our black-box ILP solver. When the algorithm needs to solve an ILP problem, the corresponding data is retrieved from the DBMS and passed to CPLEX using tuple iterator APIs to avoid having more than one copy of the same data stored in main memory at any time. We used the same settings for all solver executions: we set its working memory to 512MB; we instructed CPLEX to store exceeding data used during the solve procedure on disk in a compressed format, rather than using the operating system's virtual memory, which, as per the documentation, may degrade the solver's performance; we instructed CPLEX to emphasize optimality versus feasibility to dampen the effect of internal heuristics that the solver may employ on particularly hard problems; we enabled CPLEX's memory emphasis parameter, which instructs the solver to conserve memory where possible; we set a solving time limit of one hour; we also made sure that the operating system would kill the solver process whenever it uses the entire available main memory. Our code is publicly available on our project website: http://packagebuilder.cs.umass.edu.

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

Datasets and queries
We demonstrate the performance of our query evaluation methods using both real-world and benchmark data. The real-world dataset consists of approximately 5.5 million tuples extracted from the Galaxy view of the Sloan Digital Sky Survey (SDSS) [34], data release 12. For the benchmark datasets we used TPC-H [35], with table sizes up to 11.8 million tuples.

We constructed a workload of seven feasible package queries for each dataset, by adapting existing SQL queries originally designed for each of the two datasets. For the Galaxy dataset, we adapted real-world sample SQL queries available directly from the SDSS website.[4] For the TPC-H dataset, we adapted seven SQL query templates provided

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

with the benchmark that contained enough numerical attributes. We performed query specification manually, by transforming SQL aggregates into global predicates or objective criteria whenever possible, selection predicates into global predicates, and by adding cardinality bounds. We did not include any base predicates in our package queries because they can always be pre-processed by running a standard SQL query over the input dataset (Section 3), and thus eliminated beforehand. For the Galaxy queries, we synthesized the global constraint bounds by multiplying the original selection bounds by the package cardinality bounds. For the TPC-H queries, we generated global constraint bounds uniformly at random by multiplying random values in the value range of a specific attribute by the cardinality bounds. We transformed the original TPC-H SQL queries into single-relation package queries by joining the original TPC-H tables using full outer joins, containing all attributes needed by all the TPC-H package queries in our benchmark. This pre-joined table contained approximately 17.5 million tuples. For each TPC-H package query, we then extracted the subset of tuples having non-NULL values on all the query attributes. The size of each resulting table is reported in Table 5. Finally, we do not allow tuple repetitions in any of the queries as they only affect the domains of the ILP integer variables. We observed that allowing tuple repetitions results in easier problems for the ILP solver.

Comparisons
We compare DIRECT with SKETCHREFINE. Both methods use the ILP formulation (Section 3) to transform package queries into ILP problems: DIRECT translates and solves the original query; SKETCHREFINE translates and solves the subqueries (Section 4), and uses *hybrid sketch query* (Section 4.2.2) as the only strategy to cope with infeasible initial queries.

Metrics
We evaluate methods on their efficiency and effectiveness.

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

Approximation ratio: Recall that SKETCHREFINE is always guaranteed to return an approximate answer with respect to DIRECT (Section 5.1). In order to assess the quality of a package returned by SKETCHREFINE, we compare its objective value with the objective value of the package returned by DIRECT on the same query. Using Obj_S and Obj_D to denote the objective values of SKETCHREFINE and DIRECT, respectively, we compute the empirical *approximation ratio* $\frac{Obj_D}{Obj_S}$ for maximization queries, and $\frac{Obj_S}{Obj_D}$ for minimization queries. An approximation ratio of one indicates that SKETCHREFINE produces a solution with same objective value as the solution produced by the solver on the entire problem. Typically, the approximation ratio is greater than or equal to one. However, since the solver employs several approximations and heuristics, values lower than one, which means that SKETCHREFINE produces a better package than DIRECT, are possible in practice.

6.2 Results and Discussion

We evaluate four fundamental aspects of our algorithms: (1) their query response time and approximation ratio with increasing dataset sizes; (2) the impact of varying partitioning size thresholds, τ, on SKETCHREFINE's performance; (3) the impact of the attributes used in offline partitioning on query runtime; (4) the impact of enforcing approximation guarantees, ε, on the performance of SKETCHREFINE.

6.2.1 Query performance as dataset size increases

In our first set of experiments, we evaluate the scalability of our methods on input relations of increasing size. First, we partitioned each dataset using the union of all package query attributes in the workload: we refer to these partitioning attributes as the *workload attributes*. We did not enforce 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. Because we do not enforce approximation guarantees, the group centroids are used as representatives for all queries. In Section 6.2.4, we specifically test how varying the diameter requirements through ε affects the running time of SKETCHREFINE.

We perform offline partitioning setting the partition size threshold τ to 10% of the dataset size. Table 6 reports the partitioning times for the two datasets. We derive the partitionings for the smaller data sizes (less than 100% of the dataset) in the experiments, by randomly removing tuples from the original partitions. This operation is guaranteed to maintain the size condition.

Figure 7 reports our scalability results on the Galaxy and TPC-H benchmarks. 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 on the galaxy dataset does not report approximation ratios, because DIRECT evaluation fails to produce a solution for this query across all

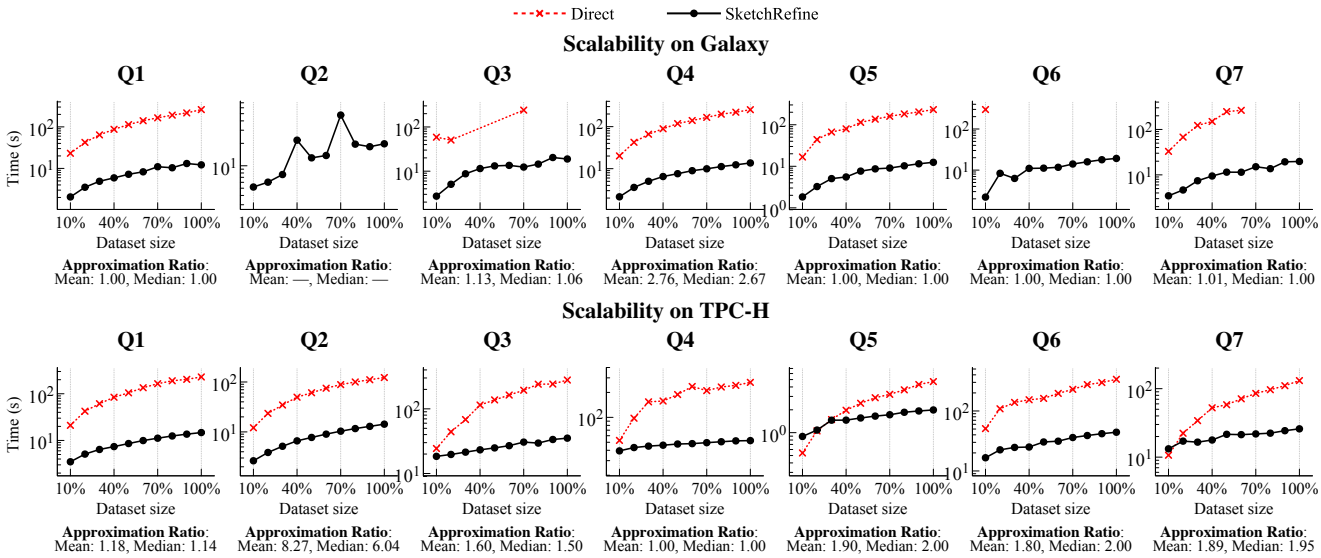

Fig. 7: Scalability on the Galaxy and TPC-H benchmarks. SketchRefine uses an offline partitioning computed on the full dataset, using the workload attributes, $\tau = 10\%$ of the dataset size, and no diameter condition. In Galaxy, Direct scales up to millions of tuples in about half of the queries, but it fails on the other half. In TPC-H, Direct scales up to millions of tuples in all queries. SketchRefine scales up nicely in all cases, and runs about an order of magnitude faster than Direct. Its approximation ratio is generally very low, even though the partitioning is constructed without diameter conditions.

data sizes. We observe that Direct can scale up to millions of tuples in three of the seven Galaxy queries, and in all of the TPC-H queries. Its run-time performance degrades, as expected, when data size increases, but even for very large datasets Direct is usually able to answer the package queries in less than a few minutes. However, Direct has high failure rate for some of the Galaxy queries, indicated by the missing data points in some graphs (queries Q2, Q3, Q6 and Q7 in the Galaxy dataset). This happens when CPLEX uses the entire available main memory while solving the corresponding ILP problems. For some queries, such as Q3 and Q7, this occurs with bigger dataset sizes. However, for queries Q2 and Q6, Direct even fails on small data. This is a clear demonstration of one of the major limitations of ILP solvers: they can fail even when the dataset can fit in main memory, due to the complexity of the integer problem. In contrast, our scalable SketchRefine algorithm is able to perform well on all dataset sizes and across all queries. SketchRefine consistently performs about an order of magnitude faster than Direct across all queries, both on real-world data and benchmark data. Its running time is consistently below one or two minutes, even when constructing packages from millions of tuples.

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

6.2.2 Effect of varying partition size threshold

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

Our results show that the partition size threshold has a major impact on the execution time of SketchRefine, with

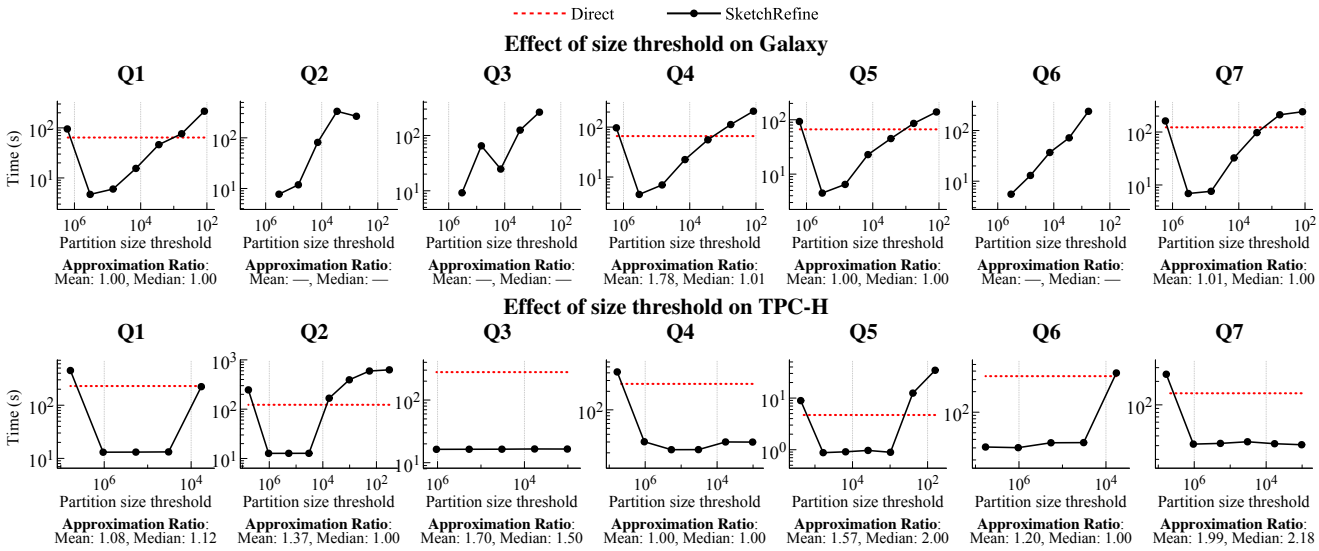

Fig. 8: Impact of partition size threshold τ on the Galaxy and TPC-H benchmarks, using, respectively, 30% and 100% of the dataset. Partitioning is performed at each value of τ using all the workload attributes, and with no diameter condition. The baseline DIRECT and the approximation ratios are only shown when DIRECT is successful. The results show that τ has a major impact on the running time of SKETCHREFINE, but almost no impact on the approximation ratio. SKETCHREFINE can be an order of magnitude faster than DIRECT with proper tuning of τ.

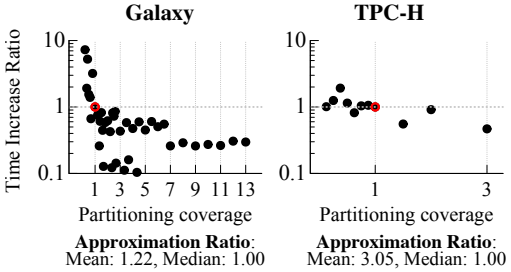

Fig. 9: Increase or decrease ratio in running time of SKETCHREFINE with different partitioning coverages. Coverage one, shown by the red dot, is obtained by partitioning on the query attributes. The results show an improvement in running time when partitioning is performed on supersets of the query attributes,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

extreme values of τ (either too low or too high) often resulting in slower running times than DIRECT. With bigger partitions, on the left-hand side of the *x*-axis, SKETCHREFINE takes about the same time as DIRECT because both algorithms solve problems of comparable size. When the size of each partition starts to decrease, moving from left to right on the *x*-axis, the response time of SKETCHREFINE decreases rapidly, reaching about an order of magnitude improvement with respect to DIRECT. Most of the queries show that there is a "sweet spot" at which the response time is the lowest: when all partitions are small, and there are not too many of them. The point is consistent across different queries, showing that it only depends on the input data size (refer to Table 5 for the different TPC-H data sizes). After that point, although the partitions become smaller, the number of partitions starts to increase significantly. This increase has two negative effects: it increases the number of representative tuples, and thus the size and complexity of the initial SKETCH query, and it increases the number of groups that REFINE may need to refine to construct the final package. This causes the running time of SKETCHREFINE, on the right-hand side of the *x*-axis, to increase again and reach or surpass the running time of DIRECT. We only report mean and median approximation ratios, which are in all cases very close to one, indicating that SKETCHREFINE retains very good quality regardless of the partition size threshold. We studied how different partitioning size thresholds (τ) affect approximation ratios. We observed that the ratio follows an inverse trend to that of the running time in Figure 8. In the two extreme cases, when there is only one partition of size *n* (SKETCHREFINE is a single refine query that corresponds to DIRECT) and when there are *n* partitions of size 1 (SKETCHREFINE is a sketch query over *n* groups of a single tuple each), SKETCHREFINE returns the optimal solution (approximation ratio 1). Between these endpoints, for some queries, the approximation ratio can be higher than 1. With a smaller number of partitions, our partitioning algorithm produces larger partitions with potentially large diameters, but each refine query produces an optimal solution over a larger subproblem. As the number of partitions increases, the refine query operates over smaller subproblems leading to worse approximation ratios, until

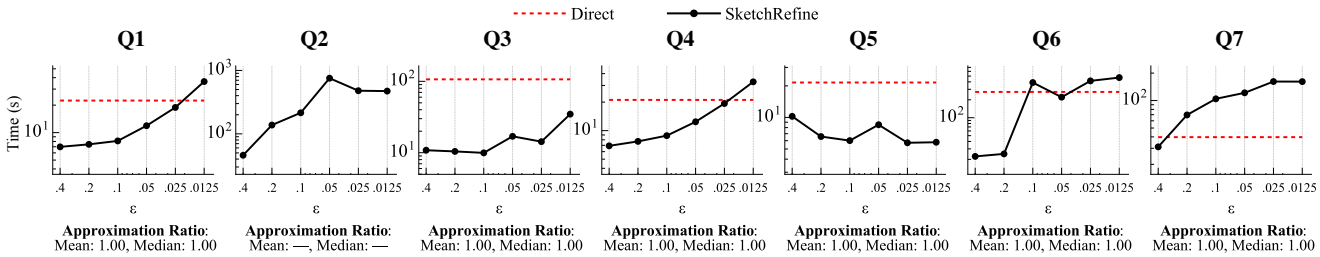

Fig. 10: Impact of the approximation parameter ε (increasingly stricter approximation requirements) on the Galaxy workload, using 10% of the dataset size. Partitioning is performed on the query attributes, without enforcing a limit on the size of the partitions, τ, while imposing diameter limits governed by ε. The baseline DIRECT and the approximation ratios are only shown when DIRECT is successful. The results show that ε has a major impact on the running time of SKETCHREFINE, as a smaller ε implies smaller partition diameters and, thus, more partitions, while maintaining the approximation ratio always down to 1.

the partitions start to have tighter diameters leading to better approximation.

6.2.3 Effect of varying partitioning coverage

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

For each query, we report the effect of the partitioning coverage on query runtime as the ratio of a query response time over the same query's response time when coverage is one: a higher ratio (> 1) indicates slower response time and a lower ratio (< 1) indicates a faster response time. Figure 9 reports the results on the Galaxy and the TPC-H datasets. The Galaxy dataset has many more numerical attributes than the TPC-H dataset, allowing us to experiment with higher values of coverage. The response time of SKETCHREFINE improves on both datasets when the offline partitioning covers a superset of the query attributes, whereas it tends to increase when it only considers a subset of the query attributes. The mean and median approximation ratios are consistently low, indicating that the quality of the packages returned by SKETCHREFINE remains unaffected by the partitioning coverage.

These results demonstrate that SKETCHREFINE is robust to imperfect partitioning, which do not cater precisely to the query attributes. Moreover, using a partitioning over a superset of a query's attributes typically leads to better performance. The reason for this is twofold: First, higher coverage achieves partitioning groups where tuples are similar across all attributes pertinent to the query. Thus, the sketch query uses better representatives and produces a more relevant initial package, and the refine queries are more likely feasible. Second, partitioning on more attributes can also achieve smaller partitioning groups. As a result, this speeds up the refine queries, and also reduces the diameter of each group, with the potential of improving the approximation ratio. This means that partitioning can be performed offline using the union of the attributes of an anticipated workload, or even using all the attributes of a relation.

6.2.4 Effect of varying ε

In our final set of experiments, we study the impact of different approximation guarantees on the query response time and the approximation ratio of SKETCHREFINE. We vary ε, the approximation parameter, from higher values (looser approximation bound) to lower values (tighter approximation bound), and enforce diameter limits according to Theorem 3. A looser approximation bound can cause the algorithm to produce package results with a worse objective value. More specifically, $\varepsilon = 0.4$ guarantees approximation ratios not worse than 1.4 for minimization queries and 1.67 for maximization queries, and $\varepsilon = 0.0125$ guarantees approximation ratios not worse than 1.0125 for minimization queries and 1.0127 for maximization queries. Figure 10 presents the results on the Galaxy workload, where ε varies from high values, on the left-hand size of the x-axis, to lower values. When DIRECT is able to produce a solution, we also report its running time (horizontal line) as a baseline for comparison.

Enforcing stricter (lower) ε leads to an increase in the running time of SKETCHREFINE. This is expected, as the stricter diameter bounds result in more partitions, and as we observed in our partition threshold experiments (Section 6.2.2), having more partitions can negatively impact the running time of SKETCHREFINE. This trade-off between quality and runtime performance is a known characteristic of most approximation schemes [36].

Our results also show that enforcing even a loose ε, such as 0.4, enables SKETCHREFINE to compute a result to all the queries faster than DIRECT with no cost in quality, as the

observed approximation ratios are always equal to 1. Notably, this happens in all the queries, including those that showed higher approximation ratio in the previous experiments where the approximation guarantee was not enforced.

7 Parallelizing SKETCHREFINE

Our evaluation showed that SKETCHREFINE outperforms DIRECT on both the Galaxy and the TPC-H datasets. Specifically, SKETCHREFINE has three important advantages: First, it scales naturally to very large datasets, by breaking down the problem into smaller, manageable subproblems, whose solutions can be combined to form the final result. Second, it provides flexible approximations with strong theoretical guarantees on the quality of the package results. Third, while our current implementation employs ILP solvers, SKETCHREFINE can use any arbitrary black-box algorithm to evaluate the generated package subproblems, even solutions that work entirely in main memory [16, 36, 14], and whose efficiency drastically degrades with larger problem sizes. SKETCHREFINE will offer the same efficiency gains and approximation guarantees over the employed black-box algorithm.

However, there are two scenarios that can degrade SKETCHREFINE's performance. First, as we discussed in Section 4.2.2, the worst-case running time of the algorithm is exponential in the number of partitions, due to the backtracking logic in the REFINE phase. The REFINE algorithm may get caught in a sequence of promising refine orderings that fail at their last step. Our evaluation showed that this scenario is uncommon in practice, and the algorithm was always able to quickly find a successful refine order for the partitioning groups. Second, SKETCHREFINE achieves most of its gains in the SKETCH phase, which identifies the relevant partitions, reducing the work of REFINE. Thus, the algorithm is susceptible to bad performance when queries require tuples to be picked from a large number of the partitions. We investigate this scenario in more detail, starting with a motivating example from the Galaxy dataset.

Example 6 (Varied Red Galaxies) Similar to Example 2, an astrophysicist is looking for rectangular regions of the night sky that may contain previously unseen celestial objects. This time, the scientist is specifically looking for galaxies that span different brightness levels on the red color component.

In this example, the astrophysicist requires each galaxy (package) to include red color components from the entire red spectrum. We can encode this in PaQL by dividing the red spectrum into ranges, and requiring the resulting package to include at least one tuple from each range interval. Each such constraint would be of the following form:

(SELECT COUNT$(*)$ FROM P
WHERE r BETWEEN r_{lb} AND r_{ub}) $>= 1$

where r is the name of the red color component from the Galaxy schema, and r_{lb} and r_{ub} are the lower and upper bounds of one of the range intervals. The query has one such constraint for each range interval. Each such constraint forces the result package to contain at least one tuple in the specified r range.

If the dataset is partitioned on the red color component, r, these constraints will force SKETCHREFINE to generate and solve a subproblem for most of the partitions, causing a substantial increase to its running time. In the worst case, REFINE will need to operate on all partition groups, and its performance can get as bad as DIRECT.

We implement the scenario of this example in our Galaxy workload by partitioning the data on attribute r, generating 14 partitioning groups. We create constraints based on range intervals that correspond to the partitioning on r. Then for each Galaxy query, we generate a sequence of 14 queries, by augmenting the query with more constraints on r, thus forcing increasingly higher partition utilization. The first query of the sequence only has one color constraint requiring at least one tuple from a single partition, corresponding to the lowest partition utilization (~10%). The last query of the sequence has 14 color constraints, one for each partition, requiring at least one tuple from *each* partition. This corresponds to the highest partition utilization (100%). Queries with more constraints on r will require the REFINE phase to solve more partitions. We observe the impact of this workload on SKETCHREFINE's performance in Figure 11: As partition utilization increases (due to more constraints on r), the runtime of greedy SKETCHREFINE increases, and matches that of DIRECT when most partitions are needed. In this experiment, the runtime of DIRECT also increases, as the addition of the partition constraints makes each query individually more complex.

Since SKETCHREFINE relies on solving several smaller subproblems, a natural way to improve its performance is by parallelizing the REFINE step. Unfortunately, the greedy backtracking algorithm (Algorithm 2) requires incremental refinements, always maintaining the feasibility of the intermediate solutions. Each step in the algorithm makes a local decision based on results of the previous decisions and their order. Thus, solving the REFINE subproblems in parallel does not guarantee that the overall package will be feasible.

In this section, we introduce a new *iterative* method for performing the REFINE phase of SKETCHREFINE. The iterative algorithm has the following advantages over Algorithm 2: (1) It allows partitions to be evaluated in parallel, independently from each other; (2) It eliminates the need for backtracking and, thus, its exponential worst-case; (3) It can reach infeasibility faster than backtracking, while offering the same false-infeasibility bounds; (4) It guarantees the same approximation bounds. Figure 11 shows that parallel execution of our new *iterative* SKETCHREFINE leads to significant

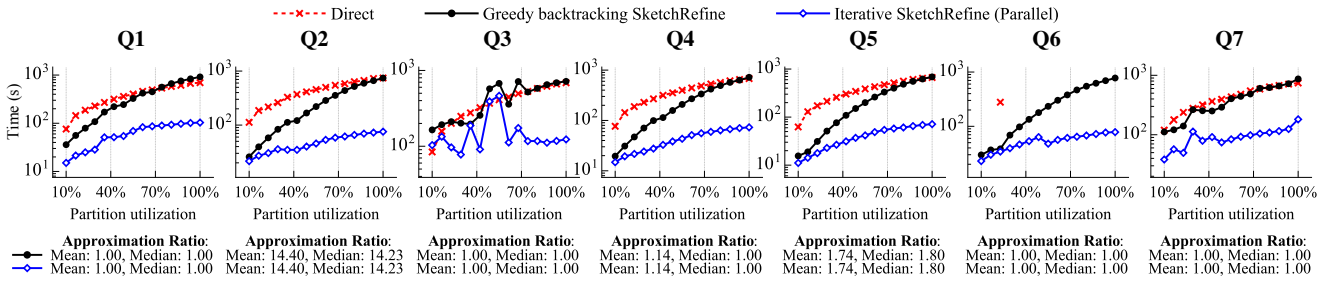

Fig. 11: Impact of increased partition utilization on SketchRefine, on 30% of the Galaxy data. Partitioning is over attribute r only, using $\tau = 10\%$ and no diameter bounds. The performance of SketchRefine degrades as partition utilization increases, approaching the runtime of Direct. The runtime of Direct also increases, as the constraints that force higher partition utilization increase the complexity of the query.

gains in performance, and avoids the degradation that greedy SketchRefine demonstrates in cases of high partition utilization. We proceed to describe the new Refine algorithm, explain how it parallelizes, and demonstrate its scalability.

7.1 Iterative Refine

The Refine step of SketchRefine processes the sketch package $p_{\mathcal{S}}$ to replace the representative tuples with tuples from each partition. It does this by defining and solving appropriate ILP problems within each partition (refinement). The greedy backtracking implementation of Refine (Algorithm 2) performs refinements one at a time, and requires each refinement to yield a feasible package for the original query; if a refinement does not, the algorithm backtracks. We now present an alternative strategy for the Refine step that relaxes this requirement until a tuple solution for every partitioning group is found. Specifically, *iterative* Refine performs refinements independently on each partition, modifying the sketch package based on all the successful refinements, and repeating any failed ones using the new revised sketch package. Only after all partitions are solved, the algorithm ensures that feasibility of the resulting package. Algorithm 3 details the procedure, which works in two phases:

Phase 1: Iterative refinements

The first phase of the algorithm (lines 2–19) performs refinements on all unsolved partitions ($\mathcal{S}$) iteratively. At each iteration, for each unsolved partition, the algorithm solves an ILP (constructed as in Section 4.2.2) to replace the representative tuples with tuples from the partition. The algorithm updates the sketch package ($p_{\mathcal{S}}$) based on the refinements (line 15). This process repeats while there are still unsolved partitions, i.e., partitions that failed to produce a feasible solution in previous iterations (line 3). The refinement queries $\mathcal{Q}_i(p_{\mathcal{S}})$ in the new iterations will be different from their earlier versions, as the constraints on each refine query depend on $p_{\mathcal{S}}$, which has been modified by the previous iterations. Phase 1 fails (line 19) if, during an iteration, none of the partition groups can be solved: In this case, $\mathcal{S}$ remains unchanged, and the algorithm cannot make progress towards the completion of the package.

During this phase, the algorithm does not check whether $p_{\mathcal{S}}$ is a feasible solution to the overall query. Rather, the objective of this phase is to produce feasible solutions for *each* of the partition groups.

Algorithm 3 Iterative Refine

input:
- $\mathcal{Q}$: the package query to be evaluated
- $\mathcal{P} = \{(G_1, \tilde{t}_1), \ldots, (G_m, \tilde{t}_m)\}$: partitioning groups
- $\mathcal{S}$: partitioning groups yet to be refined (initially $\mathcal{S} = \mathcal{P}$)
- $p_{\mathcal{S}}$: the refining package (initially the result of Sketch)

output: a feasible package containing only tuples, or failure

```
1:  procedure REFINE(Q, P, p_S)
2:      ▷ Phase 1: Iterative refinements
3:      while S ≠ ∅ do
4:          ▷ Solve all unsolved groups S independently
5:          S' = S
6:          for all (G_i, t̃_i) ∈ S do
7:              ▷ Skip groups that have no representative in p_S
8:              if t̃_i ∈ p_S then
9:                  p_i ← DIRECT(Q_i(p_S))
10:                 if Q_i(p_S) is feasible then
11:                     S' = S' \ {(G_i, t̃_i)}
12:         if S' ⊂ S then
13:             ▷ Combine independent solutions into p_S
14:             for all (G_i, t̃_i) ∈ S do
15:                 p_S ← p_S \ {t̃_i} ∪ p_i
16:             S ← S'
17:         else if S' = S then
18:             ▷ No progress could be made
19:             return failure
20:     ▷ Phase 2: Feasibility adjustment
21:     if p_S is infeasible for Q then
22:         ▷ Attempt re-refining groups having at least one tuple
23:         for all (G_i, t̃_i) ∈ P s.t. p_S ∩ G_i ≠ ∅ do
24:             p_i ← DIRECT(Q_i(p_S), p_S)
25:             if Q_i(p_S) is feasible then
26:                 p ← p_S \ {t̃_i} ∪ p_i   ▷ Invariant: p is feasible for Q
27:                 return p
28:         return failure
29:     return p_S
```

Phase 2: Feasibility adjustment
If Phase 1 concludes successfully, the algorithm enters Phase 2 (lines 20–29) to verify whether p_S is a feasible solution to the overall query and attempt a correction if it is not. If p_S is not a feasible solution, the algorithm tries one more refine round of all partitions based on the current p_S. If any of the refine queries succeeds in this round, then the new, refined p_S is guaranteed to be a feasible solution and the algorithm returns it. If all refinement queries are infeasible, the algorithm fails. Thus, iterative REFINE may fail in two cases: (1) if all refining queries fail in one iteration of Phase 1; or (2) if the refined sketch package p_S is infeasible and unfixable in Phase 2.

Run time complexity
We denote with $T(\tau)$ the time taken by the solver to solve a problem of size τ, and express the time complexity of the refine procedure as a function of $T(\tau)$ and m, the number of partitioning groups. The best case for Algorithm 3 is that all refine queries succeed in the first iteration of Phase 1 and, in Phase 2, the refined p_S is already a feasible solution. In this case, the algorithm makes up to m calls to the solver. In the worst case, only one refine query succeeds in each iteration of Phase 1, the refined package p_S is not a feasible solution to the overall query, and only the last attempt of Phase 2 succeeds in rendering p_S feasible. In this case, Algorithm 3 makes up to $\frac{m(m+1)}{2} + m$ calls to the solver ($O(T(\tau)m^2)$).

Comparison with greedy backtracking REFINE
However, in sequential settings greedy backtracking can outperform iterative REFINE in practice. Specifically, if the subproblems can be solved independently of each other, but fail when combined, iterative REFINE requires extra steps in Phase 2 to adjust the solution. On the other hand, greedy backtracking would terminate as soon as all subproblems are solved, as it always maintains feasible solutions. With infeasible groups, iterative REFINE may also require several Phase 1 iterations, while greedy backtracking would immediately backtrack at the first infeasible group. This means that Algorithm 2 is likely to beat Algorithm 3 in harder problems, which have few feasible solutions.

7.2 Parallelizing iterative REFINE

Iterative REFINE is naturally amenable to parallelization, since all refinement problems are solved independently from each other. In particular, during Phase 1, the algorithm solves groups independently without ensuring the feasibility of the overall package. Therefore, all the refine queries in each iteration of Phase 1 can be solved in parallel, and their solutions can be combined by a central node at the end of every iteration. During Phase 2, all the refine queries are also independent because the algorithm can stop if *any* of them succeeds. Thus, all refine queries of Phase 2 can also be executed in parallel, and if any succeeds, the other ones can be immediately terminated. A central node dispatches the refine queries to be solved at each iteration to the parallel worker nodes, and combines their result into p_S, the refining sketch package. Thus, every worker node is responsible for a different partitioning group. If there are more partitioning groups than workers, the load can be easily balanced among the workers by assigning them to an equal number of groups.

7.3 Experimental evaluation of parallel SKETCHREFINE

We evaluate the scalability and effectiveness of parallel SKETCHREFINE using a variation of the queries of our Galaxy workload based on Example 6. Specifically, we partition our data on the red color component attribute, r, with $\tau =$ 10% of the original dataset size and no diameter conditions, and we modify the Galaxy queries to include cardinality constraints on ranges of r. Our partitioning on r generates 14 groups, and the runtime improvements that we report in this section are achievable with 14 parallel worker nodes (one for each partitioning group). In each experiment, we measure the running time and the approximation ratio (described in Section 6.1) of the algorithms for increasing dataset sizes, comparing DIRECT with two versions of SKETCHREFINE: one that uses greedy backtracking REFINE (Algorithm 2), and one that uses iterative REFINE (Algorithm 3).

In our first experiment, we change the number of cardinality constraints on ranges of r for each query: the more constraints, the more partitions SKETCHREFINE will need to explore. As we have seen, the performance of SKETCHREFINE with greedy backtracking degrades as partition utilization increases (Figure 11). In contrast, we observe that parallel iterative SKETCHREFINE maintains consistently better performance than DIRECT.

For our second experiment, we pick the query workload with the highest partition utilization (100%), which requires *all* of the partitions to be refined. Figure 12 reports the results. All queries show similar performance because they all share the same 14 cardinality constraints on r. Both of the SKETCHREFINE versions scale to millions of tuples, whereas DIRECT fails in many of the queries when the dataset gets too big. Here, DIRECT fails for the same reasons as our earlier experiments in Section 6.2.1. In all the cases in which DIRECT succeeds, as the dataset size increases, greedy backtracking SKETCHREFINE shows the same run-time performance as DIRECT. In fact, requiring galaxies that span all of the red color ranges requires tuples to be picked from each partition, which corresponds to the worst case for greedy backtracking. On the other hand, parallel SKETCHREFINE is able to always find an answer in about an order of magnitude less time than greedy backtracking and DIRECT. This happens because the algorithm is able to parallelize all the necessary refinements.

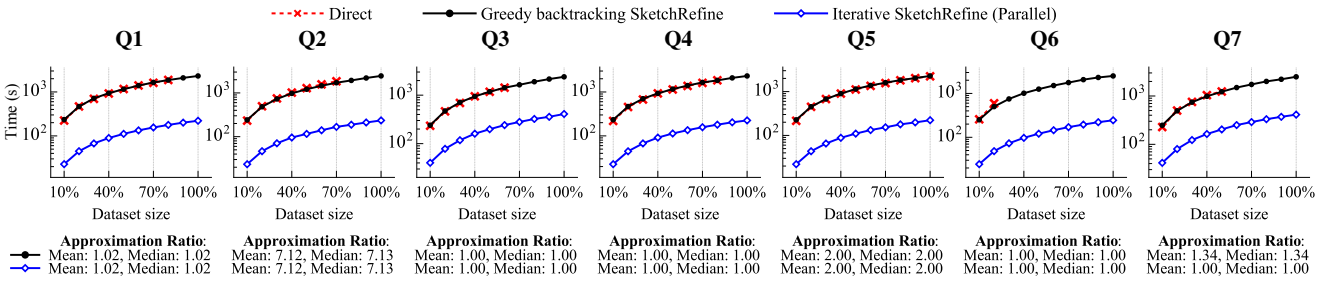

Fig. 12: Scalability of parallel SKETCHREFINE compared to greedy backtracking SKETCHREFINE and DIRECT on the *varied red* Galaxy workload. SKETCHREFINE uses partitioning computed on attribute r, $\tau = 10\%$, and no diameter condition. The running time of greedy backtracking SKETCHREFINE and DIRECT are equal as all partitions need to be refined (worst case of greedy backtracking). Parallel SKETCHREFINE scales up to nicely in all cases, and runs about an order of magnitude faster than both DIRECT and greedy backtracking. The approximation ratios of the two algorithms are both generally low, even though the partitioning is constructed without quality guarantees in place.

In this set of experiments, we did not enforce approximation guarantees, so the algorithms can potentially produce bad solutions. However, our results show that this happens rarely, and the approximation ratios of both of the SKETCHREFINE algorithms are generally very low (close to one). One exception is query Q2, for which SKETCHREFINE produces a 7-factor approximation. Finally, the approximation quality of parallel iterative SKETCHREFINE is equal (queries Q1–Q6) or better (Q7) than greedy backtracking. This shows that the gains obtained by parallelizing SKETCHREFINE do not come at the cost of quality and, in some cases, can also produce better solutions.

8 Incremental package evaluation

At the core of our package evaluation methods, DIRECT is used as a black-box evaluation strategy to solve each subproblem. Treating the subproblem evaluation as a black box is a powerful abstraction: it allows our SKETCHREFINE strategies to benefit from using alternative evaluation algorithms at this core, while the results of our theoretical analysis still hold (Section 5). In this section, we explore the potential of improving the performance of DIRECT directly, thus, slightly "lifting the lid" on this black box and exploiting some of its logic. Specifically, we will study the impact of *preconditioning*, i.e., an initial assignment of the variables, to the ILP solver's performance. The intuition is that providing the solver with a "good" starting package can reduce the search space and allow the solver to reach a solution faster.

In this paper, we do not present a particular method for identifying appropriate starting packages; our goal is to evaluate through a preliminary empirical analysis whether such a method can improve the efficiency of package evaluation in a meaningful way. Our analysis explores the following questions: (1) How does a starting package solution impact the runtime of DIRECT? (2) Does the feasibility of the starting package make a difference?

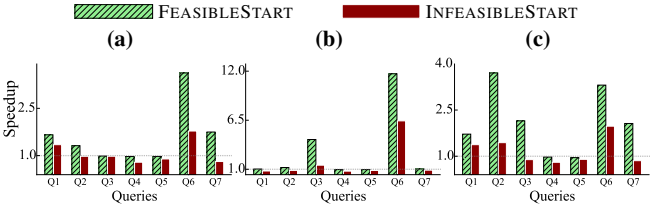

Fig. 13: Average speedup provided by FEASIBLESTART (green bars) and INFEASIBLESTART (red bars) compared to NOSTART, across different query sequences and different methodologies for sequence creation. (a) All constraints, fixed μ; (b) One constraint, fixed μ;(c) Random constraints, random μ. The results show that starting packages do not always improve the performance, but feasible starting packages generally offer better speedup than infeasible ones.

We evaluate the effect of seeding the solver with two types of starting solutions: packages that already satisfy the query's constraints (feasible), and packages that do not (infeasible). We use the Galaxy workload to construct sequences of queries with increasing strictness. Given a query $\mathcal{Q}$, we construct the sequence $(\mathcal{Q}_1, \ldots, \mathcal{Q}_r)$, such that $\mathcal{Q}_{i+1}$ has stricter constraints than $\mathcal{Q}_i$: if $\mathcal{Q}_i$ has a constraint $\mathsf{SUM}(\mathsf{attr}) \geq 2$ and the optimal solution to $\mathcal{Q}_i$ has a value 2.2 for this sum, then this constraint for $\mathcal{Q}_{i+1}$ becomes $\mathsf{SUM}(\mathsf{attr}) \geq 2.2 + \mu$, for a small constant $\mu > 0$. We construct three sequences for each query as follows:

1. we modify *all* constraints at every step of the sequence with a fixed μ,
2. we modify *only one* constraint at a time with a fixed μ,
3. we modify a random set of constraints with a random μ.

For all sequences $(\mathcal{Q}_1, \ldots, \mathcal{Q}_r)$, the solution for $\mathcal{Q}_i$ is feasible for $\mathcal{Q}_{i-1}$, but the solution for $\mathcal{Q}_{i-1}$ is not feasible for $\mathcal{Q}_i$. We construct sequences of length up to 20 for each of the 7 queries in the Galaxy workload. Sequences can have

fewer than 20 queries if constraint changes cause a query to become infeasible. We execute each query $\mathcal{Q}_i$ in a sequence using DIRECT in three ways:

NOSTART: Providing no starting solution.

FEASIBLESTART: Preconditioning with the optimal solution to $\mathcal{Q}_{i+1}$, which is a *feasible* package for $\mathcal{Q}_i$.

INFEASIBLESTART: Preconditioning with the optimal solution to $\mathcal{Q}_{i-1}$, which is an *infeasible* package for $\mathcal{Q}_i$.

We measure the runtime *speedup* of preconditioning as the ratio of the running time of NOSTART over FEASIBLESTART and INFEASIBLESTART, for feasible and infeasible starting packages, respectively. A speedup of 1 means that preconditioning has no effect on the running time. A speedup < 1 means that preconditioning led to worse performance and a speedup > 1 means that preconditioning improved the performance. Figure 13 shows the average speedup of FEASIBLESTART and INFEASIBLESTART across each query sequence, for the three types of generated sequences. Our results show that preconditioning does not consistently improve the performance of all queries. In fact, seeding the solver with an infeasible package can frequently lead to worse performance. On the other hand, FEASIBLESTART rarely hurts runtime performance, and can often help significantly—as much as 12x improvement in our experiment. This contrast between FEASIBLESTART and INFEASIBLESTART is intuitive: DIRECT needs to derive a solution that is (1) feasible and (2) has optimal objective value, so a seed that already satisfies the first condition is more likely to be useful.

Overall, the results of our empirical analysis indicate that preconditioning is a promising strategy for improving package query performance that merits further study. We offer additional discussion on this research direction in Section 10.

9 Related work

Package recommendations. Package or set-based recommendation systems [37,38] are closely related to package queries. A package recommendation system presents users with interesting sets of items that satisfy some global conditions. These systems are usually driven by specific application scenarios. For instance, in the CourseRank [31] 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* [26,2], and recommendation of *vacation* and *travel packages* [9]. 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 [10], where the authors show that the data complexity of computing top-k packages [39] with a conjunctive query language is $\mathrm{FP}^{\mathrm{NP}}$-complete.

Semantic window queries. Packages are also related to *semantic windows* [21]. A semantic window defines a contiguous subset of a grid-partitioned space with certain global properties. For instance, astronomers can partition the night sky into a grid, and look for regions of the sky whose overal brightness is above a specific threshold. If the grid cells are precomputed and stored into an input relation, these queries can be expressed in PaQL by adding a global constraint (besides the brightness requirement) that ensures that all cells in a package must form a contiguous region in the grid space. Packages, however, are more general than semantic windows because they allow regions to be non-contiguous, or to contain gaps. Moreover, package queries also allow optimization criteria, which are not expressible in semantic window queries. A recent extension to methods for answering semantic window queries is Searchlight [22], which expresses these queries in the form of constraint programs. Searchlight uses in-memory synopses to quickly estimate aggregate values of contiguous regions. However, it does not support synopses for non-contiguous regions, and thus it cannot solve arbitrary package queries.

Iceberg queries. Iceberg queries are SQL group-by aggregation queries with a highly selective HAVING clause [12, 29,25]. Package queries are much more powerful than iceberg queries, which cannot return packages of items, (they can only return group-by aggregates), and cannot express optimization objectives.

How-to queries. Package queries are related to how-to queries [27], 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.

Answer set programming. In answer set programming (ASP) [5,14], logic programs follow a Datalog-like syntax with extended functionalities. ASP, extended with arithmetic, is able to express package queries, and packages can be seen as stable models of ASP programs. While ASP can express packages, SQL-based PaQL offers a more natural extension for most relational systems. More importantly, state-of-the-art ASP solvers, like Clingo [14] from the Potassco bundle, are not yet able to scale package computation to reasonable data sizes. We observed these shortcomings by running ASP problems for our Galaxy queries: the ASP solver did not scale to more than a few dozens of tuples, while ILP solvers scale up to millions of tuples.

Constraint query languages. The principal idea of constraint query languages (CQL) [23] is that a tuple can be generalized as a conjunction of constraints over variables. This principle is very general and creates connections between declarative database languages and constraint programming. However, prior work focused on expressing constraints over tuple values, rather than over sets of tuples. In this light, PaQL follows a similar approach to CQL by embedding in a declarative query language methods that handle higher-order constraints. However, our package query engine design allows for the direct use of ILP solvers as black-box components, automatically transforming problems and solutions from one domain to the other. In contrast, CQL needs to appropriately adapt the algorithms themselves between the two domains, and existing literature does not provide this adaptation for the constraint types in PaQL.

ILP approximations. There exists a large body of research in approximation algorithms for problems that can be modeled as integer linear programs. A typical approach is *linear programming relaxation* [36] 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* [16]. All primal-dual algorithms, however, need to keep track of all primal and dual variables and the coefficient matrix, which means that none of these methods can be employed on large datasets. On the other hand, rounding techniques and primal-dual algorithms could potentially benefit from the SKETCHREFINE algorithm to break down their complexity on very large datasets.

Approximations to subclasses of package queries. Like package queries, *optimization under parametric aggregation constraints* (OPAC) queries [17] can construct sets of tuples that collectively satisfy summation constraints. However, existing solutions to OPAC queries have several shortcomings: (1) they do not handle tuple repetitions; (2) they only address *multi-attribute knapsack queries*, a subclass of package queries where all global constraints are of the form $\mathsf{SUM}() \leq c$, with a $\mathsf{MAXIMIZE\ SUM}()$ objective criterion; (3) they may return infeasible packages; (4) they are conceptually different from SKETCHREFINE, as they generate approximate solutions in a pre-processing step, and packages are simply retrieved at query time using a multi-dimensional index. In contrast, SKETCHREFINE does not require pre-computation of packages. Package queries also encompass *submodular* optimization queries, whose recent approximate solutions use greedy distributed algorithms [28].

10 Conclusions and discussion

In this paper, we introduced a complete system that supports the specification and efficient evaluation of package queries. We presented PaQL, a declarative extension to SQL, and theoretically established its expressiveness, and we developed a flexible approximation method, with strong theoretical guarantees, for the evaluation of PaQL queries on large-scale datasets. Our experiments on real-world and benchmark data demonstrate that our scalable evaluation strategy is effective and efficient over varied data sizes and query workloads, and remains robust under suboptimal conditions, parameter settings, and queries that require most of the partitions to be accessed at query time. We extended our SKETCHREFINE method to allow for effective parallelization, and we demonstrated that it maintains good performance in adverse query scenarios. Finally, we presented an empirical study showing promise for using preconditioning to support incremental package evaluation. We proceed to discuss some potential research directions in package query evaluation.

Handling joins. In this paper we assumed that, in the presence of joins, the system simply evaluates and materializes the join result before applying the package-specific transformations. However, the materialization of the join result is not always necessary: DIRECT generates variables through a single sequential scan of the join result, and thus the join tuples can be pipelined into the ILP generation without being materialized. However, not materializing the join results means that some of the join tuples will need to be recomputed to populate the solution package. Therefore, there is a space-time trade-off in the consideration of materializing the join. Further, this trade-off can be improved with hybrid, system-level solutions, such as storing the record IDs of joining tuples to enable faster access during package generation.

Incremental evaluation. Our empirical study of preconditioning (Section 8) indicates that providing feasible packages as starting solutions can significantly speed up the computation of DIRECT. A system could take advantage of this in several ways. First, the system can maintain results of past queries in a solution pool that can be searched to identify good candidate starting packages for newly submitted queries. Second, it may be possible to construct simple feasible packages by executing a simplified package query, or even a set of traditional SQL queries. Furthermore, incremental evaluation can also directly benefit iterative query refinement (such as in data exploration), as results to previous queries are natural starting packages for subsequent ones.

Top-k package queries. In this paper, we focused on producing the single optimal result for a package query with an optimization objective. Our algorithms, DIRECT and SKETCHREFINE, are not designed to efficiently produce top-k packages, as ILP solvers typically return one solution. A naïve way of producing top-k results is to return one result at a time,

and modify the query in each iteration, so as to exclude the previous result. However, such an approach is inefficient. Efficient top-k packages is an important and interesting research direction, which may benefit from solver-specific solutions.

Acknowledgements This material is based upon work supported by the National Science Foundation under grants IIS-1420941, IIS-1421322, and IIS-1453543. This is a pre-print of an article published in the VLDB Journal. The final authenticated version is available online at: https://doi.org/10.1007/s00778-017-0483-4.

References

1. Senjuti Basu Roy, Sihem Amer-Yahia, Ashish Chawla, Gautam Das, and Cong Yu. Constructing and exploring composite items. In *SIGMOD*, pages 843–854, 2010.
2. Adil Baykasoglu, Turkay Dereli, and Sena Das. Project team selection using fuzzy optimization approach. *Cybernetic Systems*, 38(2):155–185, 2007.
3. Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
4. Johannes Bisschop. *AIMMS Optimization Modeling*. Paragon Decision Technology, 2006.
5. Piero Bonatti, Francesco Calimeri, Nicola Leone, and Francesco Ricca. A 25-year perspective on logic programming. chapter Answer Set Programming, pages 159–182. Springer-Verlag, Berlin, Heidelberg, 2010.
6. Matteo Brucato, Juan Felipe Beltran, Azza Abouzied, and Alexandra Meliou. Scalable package queries in relational database systems. *PVLDB*, 9(7):576–587, 2016.
7. Matteo Brucato, Rahul Ramakrishna, Azza Abouzied, and Alexandra Meliou. PackageBuilder: From tuples to packages. *PVLDB*, 7(13):1593–1596, 2014.
8. William Cook and M Hartmann. On the complexity of branch and cut methods for the traveling salesman problem. *Polyhedral Combinatorics*, 1:75–82, 1990.
9. Munmun De Choudhury, Moran Feldman, Sihem Amer-Yahia, Nadav Golbandi, Ronny Lempel, and Cong Yu. Automatic construction of travel itineraries using social breadcrumbs. In *HyperText*, pages 35–44, 2010.
10. Ting Deng, Wenfei Fan, and Floris Geerts. On the complexity of package recommendation problems. In *PODS*, pages 261–272, 2012.
11. Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.
12. Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 299–310, 1998.
13. Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.
14. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. In M. Leuschel and T. Schrijvers, editors, *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, volume arXiv:1405.3694v1, 2014. Theory and Practice of Logic Programming, Online Supplement.
15. GNU Bison. https://www.gnu.org/software/bison/.
16. Michel X Goemans and David 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.
17. Sudipto Guha, Dimitrios Gunopulos, Nick Koudas, Divesh Srivastava, and Michail Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB*, pages 778–789, 2003.
18. John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Applied statistics*, pages 100–108, 1979.
19. Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.
20. IBM CPLEX Optimization Studio. http://www.ibm.com/software/commerce/optimization/cplex-optimizer/.
21. Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. Interactive data exploration using semantic windows. In *SIGMOD*, pages 505–516, 2014.
22. Alexander Kalinin, Ugur Çetintemel, and Stanley B. Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB*, 8(10):1094–1105, 2015.
23. PC Kanellakis, GM Kuper, and PZ Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 1(51):26–52, 1995.
24. Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons, 2009.
25. Marc Laporte, Noel Novelli, Rosine Cicchetti, and Lotfi Lakhal. Computing full and iceberg datacubes using partitions. In *Foundations of Intelligent Systems, 13th International Symposium, ISMIS 2002, Lyon, France, June 27-29, 2002, Proceedings*, pages 244–254, 2002.
26. Theodoros Lappas, Kun Liu, and Evimaria Terzi. Finding a team of experts in social networks. In *SIGKDD*, pages 467–476, 2009.
27. Alexandra Meliou and Dan Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.
28. Baharan Mirzasoleiman, Amin Karbasi, Rik Sarkar, and Andreas Krause. Distributed submodular maximization: Identifying representative elements in massive data. In *NIPS*, 2013.
29. Raymond T. Ng, Alan S. Wagner, and Yu Yin. Iceberg-cube computation with PC clusters. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 25–36, 2001.
30. Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.
31. Aditya G. Parameswaran, Petros Venetis, and Hector Garcia-Molina. Recommendation systems with complex constraints: A course recommendation perspective. *ACM TOIS*, 29(4):1–33, 2011.
32. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
33. Florian Pinel and Lav R. Varshney. Computational creativity for culinary recipes. In *CHI*, pages 439–442, 2014.
34. The Sloan Digital Sky Survey. http://www.sdss.org/.
35. The TPC-H Benchmark. http://www.tpc.org/tpch/.
36. David P Williamson and David B Shmoys. *The design of approximation algorithms*. Cambridge University Press, 2011.
37. Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Breaking out of the box of recommendations: from items to packages. In *Proceedings of the 2010 ACM Conference on Recommender Systems, RecSys 2010, Barcelona, Spain, September 26-30, 2010*, pages 151–158, 2010.
38. Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Composite recommendations: from items to packages. *Frontiers of Computer Science*, 6(3):264–277, 2012.
39. Min Xie, Laks V. S. Lakshmanan, and Peter T. Wood. Generating top-k packages via preference elicitation. *PVLDB*, 7(14):1941–1952, 2014.

Appendix: Additional material

In this Appendix, we provide additional supplementary material to the paper *"Package queries: Efficient and scalable computation of high-order constraints"*. We discuss ASP encoding for package queries and show an example of this encoding, we expand the discussion on hybrid SKETCH, we provide details on the queries of our experimental workload, we present experimental results that demonstrate the performance trade-off between the two REFINE algorithms, and include some additional results on our study of solver preconditioning.

1 Expressing package queries with ASP

An answer set program [5] (ASP) is given by a set of *rules* of the form `head:-body`, each of which is logically equivalent to an implication *head* $\leftarrow$ *body*. A *fact* is a rule devoid of a body, used to express input data. ASP extended with arithmetic can express package queries. Each input tuple $t_i = (id_i, v_{i1}, \ldots, v_{ik})$ is encoded as a fact of the form `tuple(`$\mathtt{id}_i$`,`$\mathtt{v}_{i1}$`,...,`$\mathtt{v}_{ik}$`)`, and the dataset of n tuples as a fact `tuples(`$\mathtt{id}_1$`,...,`$\mathtt{id}_n$`)`. The query from Example 1 can be expressed in ASP (using the Clingo system [14]) as:

```
1: multiplicity(0..1).
2: 1 { package(T, N) : multiplicity(N) } 1 :- tuples(T).
3: #sum{N,T : package(T, N) : tuples(T),
       multiplicity(N)} = 3.
4: 2 <= #sum{(Kcal*N),T : package(T, N) :
       tuple(T,Kcal,...), multiplicity(N)} <= 2.5.
5: #minimize {(SaturatedFat*N),T : package(T, N),
   tuple(T,...,SaturatedFat), multiplicity(N)}.
```

Lines 1 and 2 ensure that each tuple in an output package can either be picked (`multiplicity(1)`) or not (`multiplicity(0)`). Line 3 expresses the cardinality requirement (three recipes), line 4 the sum constraint on kcal, and line 5 the objective criterion. The benefit of this encoding is its Datalog-like declarativeness. However, as we discuss in the related work section, state-of-the-art ASP solvers are not yet capable of scaling up package computation.

2 Hybrid SKETCH

Recall, from Section 4.2.1, that the SKETCH query, $\mathcal{Q}(\tilde{\mathsf{R}})$, creates an initial package solution by selecting representative tuples that satisfy the original query constraints:

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

A REFINE query $\mathcal{Q}_i(p_{\mathcal{S}})$ for group G_i, presented in Section 4.2.2, selects tuples from G_i to replace the representatives for G_i in the current solution:

$\mathcal{Q}_i(p_{\mathcal{S}})$: SELECT PACKAGE$(*)$ AS p_i
FROM G_i REPEAT 0
WHERE G_i.gluten = 'free'
SUCH THAT
COUNT$(p_i.*)$ + **COUNT**$(\bar{p}_i.*) = 3$ AND
SUM$(p_i$.kcal) + **SUM**$(\bar{p}_i$.kcal) BETWEEN 2.0 AND 2.5
MINIMIZE SUM$(p_i$.sat_fat)

An *hybrid* SKETCH query for group G_i, $\mathcal{Q}(\tilde{\mathsf{R}})_i$, combines the SKETCH query with the REFINE query for G_i into one single query to be executed in place of the SKETCH query. The hybrid query selects tuples from G_i and, at the same time, representatives from all other groups:

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

Hybrid SKETCH is a simple heuristic that can prevent false infeasibility from occurring in three cases:

1. If the original sketch query, $\mathcal{Q}(\tilde{\mathsf{R}})$, is infeasible due to a bad representative from one of the groups, SKETCHREFINE would erroneously report the query as infeasible. The hybrid SKETCH query over that group would not use the bad representatives. It would rather try to construct a package using the actual tuples from that group. This may render the query feasible and allow SKETCHREFINE to proceed.
2. A REFINE query for a particular group may fail in a later stage. This is usually caused by choices made on previous REFINE steps. An hybrid query would try to find a solution for that group earlier on, when choices are not yet made for the other groups. This can improve the chances to find a tuple solution for that group.
3. Finally, a failing REFINE query for a particular group may be avoided altogether if no representatives get picked for that group. While the original SKETCH query could pick representatives for that group, the hybrid query may not.

3 Galaxy and TPC-H PaQL workloads

For our experiments, we constructed a workload of seven feasible package queries for each of the Galaxy and TPC-H datasets, using existing SQL queries, originally designed for these datasets. We show all the queries in our workload in Figure 14.

Galaxy PaQL workload

Q_1:
```
SELECT      PACKAGE(*) AS P
FROM        Galaxy REPEAT 0
SUCH THAT   SUM(extinction_r) ≥ 0.875
AND         SUM(r) ≤ 220
AND         COUNT(*) BETWEEN 5 AND 10
MAXIMIZE    COUNT(*)
```

Q_2:
```
SELECT      PACKAGE(*) AS P
FROM        Galaxy REPEAT 0
SUCH THAT   SUM(cmodelmag_g) BETWEEN 90 AND 190
AND         SUM(ra) BETWEEN 990 AND 1800
AND         COUNT(*) BETWEEN 5 AND 10
MINIMIZE    SUM(dec)
```

Q_3:
```
SELECT      PACKAGE(*) AS P
FROM        Galaxy REPEAT 0
SUCH THAT   SUM(cx) ≥ 0.466
AND         SUM(cy) ≤ 0.689
AND         COUNT(*) BETWEEN 5 AND 10
MINIMIZE    SUM(cx)
```

Q_4:
```
SELECT      PACKAGE(*) AS P
FROM        Galaxy REPEAT 0
SUCH THAT   SUM(petrorad_r) ≤ 180
AND         COUNT(*) BETWEEN 5 AND 10
MINIMIZE    SUM(colc_u)
```

Q_5:
```
SELECT      PACKAGE(*) AS P
FROM        Galaxy REPEAT 0
SUCH THAT   SUM(petrorad_i) ≥ 75
AND         COUNT(*) BETWEEN 5 AND 10
MINIMIZE    COUNT(*)
```

Q_6:
```
SELECT      PACKAGE(*) AS P
FROM        Galaxy REPEAT 0
SUCH THAT   SUM(petromag_i) ≥ 87.5
AND         SUM(petromag_r) ≥ 77.5
AND         SUM(i) ≥ 0.001
AND         SUM(r) ≥ 0.001
AND         SUM(g) ≥ 0.001
AND         COUNT(*) BETWEEN 5 AND 10
MINIMIZE    SUM(petromag_r)
```

Q_7:
```
SELECT      PACKAGE(*) AS P
FROM        Galaxy REPEAT 0
SUCH THAT   SUM(ra) BETWEEN 900 AND 1810
AND         SUM(dec) BETWEEN −5 AND 10
AND         SUM(r) ≤ 217.5
AND         COUNT(*) BETWEEN 5 AND 10
MAXIMIZE    SUM(r)
```

TPC-H PaQL workload

Q_1:
```
SELECT      PACKAGE(*) AS P
FROM        Tpch REPEAT 0
SUCH THAT   SUM(sum_base_price) ≤ 15M
AND         SUM(sum_disc_price) ≤ 45M
AND         SUM(sum_charge) ≤ 96M
AND         SUM(avg_qty) ≤ 50.36
AND         SUM(avg_price) ≤ 69K
AND         SUM(avg_disc) ≤ 0.11
AND         SUM(sum_qty) ≤ 78K
AND         COUNT(*) ≥ 1
MAXIMIZE    SUM(count_order)
```

Q_2:
```
SELECT      PACKAGE(*) AS P
FROM        Tpch REPEAT 0
SUCH THAT   SUM(p_size) ≤ 8
AND         COUNT(*) ≥ 1
MINIMIZE    SUM(ps_min_supplycost)
```

Q_3:
```
SELECT      PACKAGE(*) AS P
FROM        Tpch REPEAT 0
SUCH THAT   SUM(revenue) ≥ 414K
AND         COUNT(*) ≥ 1
MINIMIZE    COUNT(*)
```

Q_4:
```
SELECT      PACKAGE(*) AS P
FROM        Tpch REPEAT 0
SUCH THAT   SUM(o_totalprice) ≤ 454K
AND         SUM(o_shippriority) ≥ 0
AND         COUNT(*) ≥ 1
MINIMIZE    COUNT(*)
```

Q_5:
```
SELECT      PACKAGE(*) AS P
FROM        Tpch REPEAT 0
SUCH THAT   SUM(revenue) ≥ 720K
AND         COUNT(*) ≥ 1
MINIMIZE    COUNT(*)
```

Q_6:
```
SELECT      PACKAGE(*) AS P
FROM        Tpch REPEAT 0
SUCH THAT   SUM(revenue) ≥ 73K
AND         SUM(l_quantity) ≤ 110.95
AND         COUNT(*) ≥ 1
MINIMIZE    COUNT(*)
```

Q_7:
```
SELECT      PACKAGE(*) AS P
FROM        Tpch REPEAT 0
SUCH THAT   COUNT(*) ≤ 2.667e-6*
(SELECT COUNT(*) FROM Tpch)
AND         COUNT(*) ≥ 1
MAXIMIZE    SUM(revenue)
```

Fig. 14: PaQL queries constructed for the Galaxy and TPC-H workloads.

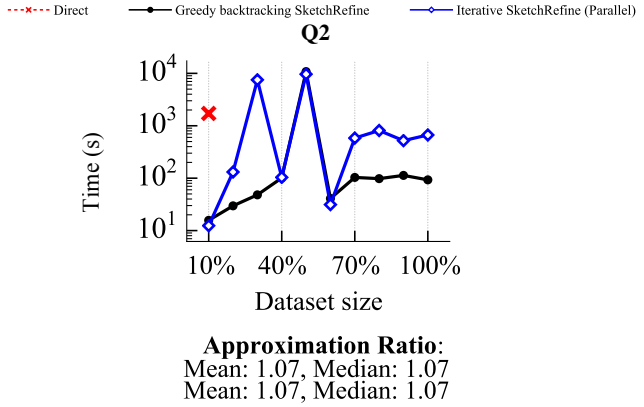

Fig. 15: Tradeoff between greedy-backtracking and iterative REFINE **on query Q_2 from the Galaxy workload, using 1.6% of the dataset size as partitioning size threshold, and without enforcing diameter conditions. The results show that greedy backtracking is faster than iterative** REFINE **in several cases.**

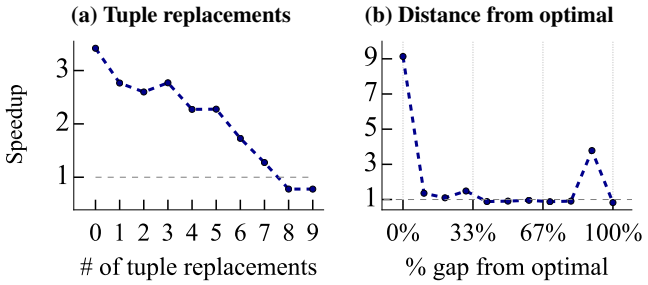

Fig. 16: Average speedup provided by starting solutions with worse objective values. Packages with tuples in common with the optimal package fare better than packages that are close to the optimal in objective value.

4 Performance trade-off between greedy-backtracking and iterative REFINE

Figure 15 shows the gains of greedy backtracking against iterative on query Q_2 from the Galaxy workload, using 1.6% of the dataset size as partitioning size threshold, and without enforcing diameter conditions. In this run, both algorithms encounter infeasible groups. Iterative REFINE requires several Phase 1 iterations to solve them, while greedy backtracking immediately backtracks at the first infeasible group. This greedy strategy proves better for greedy backtracking in most of the cases.

5 Incremental evaluation: Distance from optimal

A potential factor that could impact incremental evaluation with preconditioning is the distance of the seed package from the optimal solution. We consider two ways to measure this distance: (1) the number of tuples that differ between the packages, and (2) the difference in the objective values.

Tuple difference. For each query in the Galaxy workload, we first compute the optimal package results, and then create seed packages by replacing tuples in the optimal package. We search for replacement tuples that do not break feasibility and do not worsen the objective value by more than 1% using standard SQL. We create 10 feasible seeds for each query: $\{s_0, s_1, \ldots, s_9\}$, where s_i differs from the optimal package in i tuples; s_0 is the optimal package itself. Subsequently, s_i is created by: (1) setting s_i to s_{i-1}; (2) removing one of the tuples from s_i that also appear in s_0; (3) inserting into s_i a new tuple found via a SQL query that selects from the input table a new (unused) tuple such that, when added to s_i, generates a feasible package with an objective value not worse than the objective value of s_0 by a 1% factor. Figure 16a shows the speedup against the number of tuples replaced in the package. We see that the solver benefits a lot from seeds that share a lot of tuples with the optimal solution. As expected, the gains decrease as more tuples are replaced, and, eventually,

seeds with many different tuples can have a negative impact on performance.

Objective difference. For each query in the Galaxy workload, we create a sequence of feasible seeds $\{s_0, s_1, \ldots, s_k\}$ with progressively worse objective values. The first seed, s_0, is the optimal package itself. Subsequent seeds are generated based on a "gap" parameter $g \geq 0$: if the optimal value of s_0 is v_0, then seed s_i has objective value $v_i \geq (1 + i \cdot g)v_0$ for minimization queries and $v_i \leq (1 - i \cdot g)v_0$ for maximization queries.

Figure 16b shows the speedup when we range the gap of seed packages from 0 to 100%. The results of this experiment do not follow our initial intuition that speedup would decrease as the distance from the optimal objective value decreases. We observe that the speedup is not consistent when varying the objective value of the starting solution, and the gains reduce quickly with distance from the optimal solution. While we still observe performance gains until a distance of 30% from the optimal objective, these gains are generally small. In larger distances from the optimal, preconditioning results in some losses, but these are small as well. Contrasting these results with the experiment on tuple-based distance (Figure 16a), we conclude that it is important for a preconditioning method to target tuples that are likely to appear in the optimal package, rather than the objective value of the seed package.