Scaling Package Queries to a Billion Tuples via Hierarchical Partitioning and Customized Optimization

Anh L. Mai
New York University Abu Dhabi
anh.mai@nyu.edu

Pengyu Wang
New York University Abu Dhabi
pengyu.wang@nyu.edu

Azza Abouzied
New York University Abu Dhabi
azza@nyu.edu

Matteo Brucato
Microsoft Research
mbrucato@microsoft.com

Peter J. Haas
University of Massachusetts Amherst
phaas@cs.umass.edu

Alexandra Meliou
University of Massachusetts Amherst
ameli@cs.umass.edu

ABSTRACT

A package query returns a package—a multiset of tuples—that maximizes or minimizes a linear objective function subject to linear constraints, thereby enabling in-database decision support. Prior work has established the equivalence of package queries to Integer Linear Programs (ILPs) and developed the SketchRefine algorithm for package query processing. While this algorithm was an important first step toward supporting prescriptive analytics scalably inside a relational database, it struggles when the data size grows beyond a few hundred million tuples or when the constraints become very tight. In this paper, we present Progressive Shading, a novel algorithm for processing package queries that can scale efficiently to billions of tuples and gracefully handle tight constraints. Progressive Shading solves a sequence of optimization problems over a hierarchy of relations, each resulting from an ever-finer partitioning of the original tuples into homogeneous groups until the original relation is obtained. This strategy avoids the premature discarding of high-quality tuples that can occur with SketchRefine. Our novel partitioning scheme, Dynamic Low Variance, can handle very large relations with multiple attributes and can dynamically adapt to both concentrated and spread-out sets of attribute values, provably outperforming traditional partitioning schemes such as kd-tree. We further optimize our system by replacing our off-the-shelf optimization software with customized ILP and LP solvers, called Dual Reducer and Parallel Dual Simplex respectively, that are highly accurate and orders of magnitude faster.

PVLDB Reference Format:
Anh L. Mai, Pengyu Wang, Azza Abouzied, Matteo Brucato, Peter J. Haas, and Alexandra Meliou. Scaling Package Queries to a Billion Tuples via Hierarchical Partitioning and Customized Optimization. PVLDB, 17(5): 1146 - 1158, 2024.
doi:10.14778/3641204.3641222

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/alm818/PackageQuery.

Proceedings of the VLDB Endowment, Vol. 17, No. 5 ISSN 2150-8097.
doi:10.14778/3641204.3641222

1 INTRODUCTION

Package queries [5] extend traditional relational database queries to handle constraints that are defined over a multiset of tuples called a "package." A package has to satisfy two types of constraints:

- *Local predicates*: traditional selection predicates, i.e., constraints that each tuple in the package has to satisfy individually.
- *Global predicates*: constraints that all the tuples within the package have to satisfy collectively.

There can be many such feasible packages. A package query selects a feasible package that maximizes or minimizes a linear objective.

For example, consider the following package query: *An astrophysicist needs to find a certain number of rectangular regions of the night sky that may contain unseen quasars. These regions should have average brightness above a certain threshold and their overall red shift should lie between specified values. Among those regions, the one with the maximum combined log-likelihood of containing a quasar is preferred* [16]. Suppose that we have a `Regions` table as below:

ID	brightness	redshift	quasar	···	explored	
301	6.0	1.47	-0.05	···	true	x_1
491	9.6	1.68	-0.01	···	false	x_2
⋮	⋮	⋮	⋮	⋮	⋮	⋮

This package query can be expressed declaratively using PaQL, an SQL-based query language [5]:

```
SELECT       PACKAGE(*) AS P
FROM         Regions R REPEAT 0
WHERE        R.explored = 'false'
SUCH THAT    COUNT(P.*) = 10
             AVG(P.brightness) ≥ θ
             SUM(P.redshift) BETWEEN γ1 AND γ2
MAXIMIZE     SUM(P.quasar)
```

In this example, the number of rows, i.e., rectangular regions of the night sky, can become very large if the surveying resolution is high and/or the surveying volume of the night sky is large. For such a query, the relation size typically ranges from millions to billions of regions while the number of constraints is constant. This example shows how a package query with a very large number of rows can arise in scientific applications such as astronomy, oceanography, atmospheric science, and more. Large package queries can appear in many other domains, in the context of decision support. For example, consider a national marketing campaign where each person is exposed to one out of a possible set of k personalized ads. Each row of the table now corresponds to a (person, ad) pair. A model

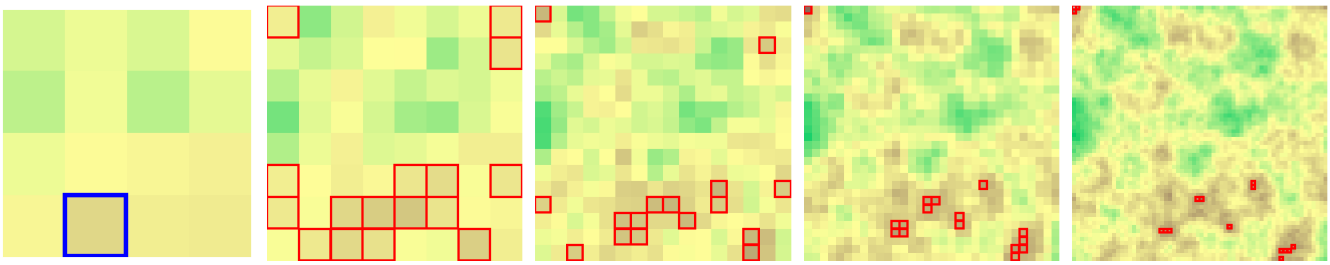

Figure 1: From left to right: five increasing resolutions of terrain height: 4x4, 8x8, 16x16, 32x32, 64x64. The blue square is the highest square in the lowest resolution. The red squares are the 16 highest squares in each subsequent resolution.

predicts the expected purchase amount for each person, given the person's features and the ad. The goal is to select an ad for each person so as to maximize predicted sales, subject to constraints on the advertising budget. The problem becomes even larger if each person can be shown multiple ads over a period of time. Other examples include certain types of portfolio optimization problems [20].

Every package query corresponds to an Integer Linear Program (ILP) [5], a common but challenging type of optimization problem. For a relation containing n tuples, there are n decision variables, with the ith decision variable x_i representing the multiplicity (possibly 0) of the ith tuple in the package. In the astrophysics example, $x_i = 1$ if the ith region is included in the package and $x_i = 0$ otherwise. Thus, setting $c_i = t_i$.quasar, we want to maximize the linear function $\sum_i c_i x_i$ subject to linear constraints such as $\sum_i x_i = 10$ and $\sum_i a_i x_i \leq \gamma_2$. where $a_i = t_i$.redshift. Thus, black-box ILP solvers like Gurobi [10] or CPLEX [30] can, in principle, be used to compute the optimal package for any package query. When n grows beyond several million, however, the foregoing solvers typically do not scale because they employ ILP techniques that have $O(\exp(n))$ worst-case running time. The SketchRefine approximate package-query processing algorithm introduced in [5] breaks down the original optimization problem into a sequence of small problems and works well up to tens of millions of decision variables. Beyond this scale, however, its performance deteriorates in both running time and optimality as shown by our experimental results in Section 4.2.Because the number of decision variables in a package query is often multiple orders of magnitude greater than the "large" problems previously studied in the optimization literature, prior approximate ILP algorithms have even more trouble scaling because, unlike SketchRefine, they need to process all the decision variables at once [9].

Overview of SketchRefine and its limitations. SketchRefine partitions relations to scalably approximate package queries. Each partition contains similar tuples that are averaged to construct a representative tuple. In SketchRefine, a "sketch" is a package solution over the representative tuples only. Representative tuples included in the sketch indicate that their groups may have tuples that can be part of the optimal package. The sketch is "refined" by searching through these groups, iteratively replacing each representative with the group's tuples, and re-solving the package-query ILP until a feasible package is constructed from the actual tuples.

SketchRefine uses kd-tree partitioning [8] with a fixed number of groups, regardless of the relation size. The number of groups is usually small (e.g., up to 1000 groups for a relation size of tens of millions) resulting in a large number of tuples in each group. While this approach allows aggressive pruning of the relation in the sketch phase, it has three drawbacks. First, the representative tuples may not accurately represent their groups, especially if the underlying distribution of tuples has a high variance. This can lead to *false infeasibility*, where no solution is found during sketching even though a feasible package does exist. Our experimental results in Section 4.2 show that the prevalence of false infeasibility increases significantly as the query constraints become tighter, i.e., as the feasible region shrinks. Second, the aggressive pruning of entire groups might eliminate potential tuples at the periphery of these groups from consideration, i.e., groups that were not selected in the sketch can contain outlying tuples that can improve the overall objective value. Without these tuples, SketchRefine can produce packages with suboptimal objective values. Third, when the relation size increases, the size of the refine queries, which essentially equals the group size, increases. It is challenging at best, and often impossible in practice, to decide exactly how fine the partitions should be. Creating too many groups will result in high computational costs for the partitioning algorithm and for solving the sketch query. Creating too few groups will degrade the accuracy of the sketch query and possibly cause false-infeasibility problems, as well as rendering the solution of each refine query hugely expensive. SketchRefine thus fails to scale to relations on the order of 100M tuples or more.

Our new approach. In this work, we introduce Progressive Shading, a novel approach for approximately solving package queries over extremely large relations which overcomes the above limitations. Our method relies on a hierarchy of relations comprising $L+1$ layers of increasingly aggregated representative tuples. Layer 0 consists of the original tuples from the relation; each layer $l > 0$ comprises representative tuples (representing groups) obtained after partitioning the tuples in layer $l-1$. Each layer comprises a large number of groups—the size of each group is small so that each representative tuple in layer l accurately summarizes the attribute values of its corresponding tuples in layer $l-1$. The search for optimal packages starts in layer L by solving a linear program (LP) over all of its representative tuples under the original constraints and objective but with the integrality requirement on the decision variables removed. The chosen tuples found in the LP are augmented with additional nearby "promising" representative tuples to help prevent premature discarding of potentially valuable tuples in layer $L-1$. Then all of the chosen tuples in layer L are expanded into their corresponding groups in layer $L-1$. This so-called Neighbor Sampling procedure of augmenting and expanding representative tuples from the current LP solution is executed at each successive

level of the hierarchy until we reach layer 0, at which point we solve a final ILP to produce the solution package (Section 2.2).

Intuitively, the differences between SketchRefine and the iterative procedure of Progressive Shading can be described via an analogy to resolution-mapping techniques in fields like geographic or demographic analysis [2]. Figure 1 shows a hierarchy of resolution from low to high of a terrain-height map that is analogous to the hierarchy of relations in Progressive Shading. An efficient approach like Progressive Shading would start from the lowest resolution and iterate toward the highest resolution. In each iteration, it starts with the 16 highest squares in the current resolution and expands those squares into the next resolution, and, among those, selects the 16 highest squares. This approach diversifies the final result in the highest resolution and thus, captures the irregularities of the terrain. On the other hand, SketchRefine is analogous to simply looking at the highest square (the blue square) in the lowest resolution and analyzing its height in the highest resolution.

The former resolution-mapping approach only works when each resolution is not downscaled too drastically to the next lower resolution—otherwise, the search within each higher-resolution square takes too long. For example, the downscale factor from 64x64 to 4x4 resolution is 256 since 5376 pixels are partitioned into 16 squares. On the other hand, going from 64x64 to 32x32 resolution has a downscale factor of 4. Analogously, the hierarchy of relations in Progressive Shading requires a partitioning algorithm that:

- Efficiently produces a large number of partitions/groups, e.g., typically about 0.1%-10% of the number of tuples (so a downscale factor between 1000 and 10). A typical kd-tree partition used in SketchRefine [5] has a downscale factor of n/g where n is the relation size and g is the fixed number of groups (at most 1000), which explains why kd-tree is not particularly suitable for Progressive Shading when n is large; and
- Supports fast group-membership determination for arbitrary tuple values (not necessarily appearing in the relation), as needed for efficient execution of Progressive Shading.

We, therefore, provide a novel partitioning algorithm, Dynamic Low Variance (DLV), that satisfies these requirements. The advantages of DLV over standard partitioning algorithms such as k-means [12], hierarchical clustering [18], and k-dimensional quadtrees [8] are (1) its ability to run under limited memory, (2) its cache-friendliness, and (3) its high parallelizability. Importantly, DLV is a *dynamic* scheme, which allows it to refine its partitions in response to outliers, i.e., to the shape of the distribution of the tuple's attributes: in our stylized example, a DLV partition on the 64x64 resolution can isolate high peaks into their own groups to maintain low variance within groups. DLV minimizes attribute variance to implicitly ensure that similar tuples are grouped together.

At the end of Progressive Shading, we end up with an in-memory ILP of a package query with tuples from the original relation. This ILP typically has at least hundreds of thousands of variables. Black-box ILP solvers would require a large amount of time to produce an optimal solution (especially when the underlying ILP is hard to solve) and hence, are unsuitable for Progressive Shading. We, therefore, develop Dual Reducer, a new heuristic algorithm that can solve a package query over millions of tuples in less than a second with close-to-optimal objective values. It

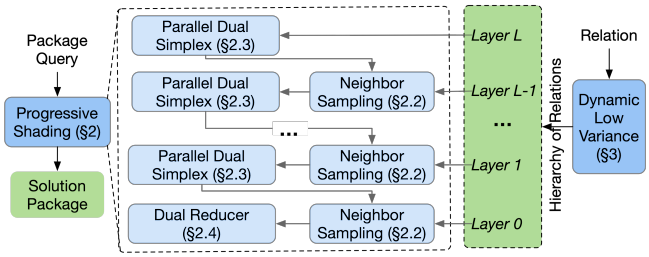

Figure 2: High-level architecture of Progressive Shading and DLV for scaling package query evaluation over very large relations.

achieves this by first solving an LP that essentially removes the integrality constraints of the ILP and then formulates a second LP using constraints that help prune tuples whose corresponding decision variables likely will not appear in the ILP solution. It effectively shrinks the original ILP into a very small sub-ILP that can be efficiently handled by black-box ILP solvers. As with any heuristic ILP solver, false infeasibility can occur in Dual Reducer when the pruning is too aggressive. We handle this issue by gradually reducing the degree of pruning until we end up solving the original ILP using a black-box ILP solver.

Progressive Shading extensively uses an LP solver for its intermediate layers and an ILP solver for layer 0. To further boost performance, we replace the intermediate black-box LP solver with our highly accurate and much faster implementation Parallel Dual Simplex, which exploits the special structure of the ILPs that arise when solving package queries compared to general ILPs (Section 2.3). We also replace the final black-box ILP solver with our novel Dual Reducer heuristic ILP solver (Section 2.4); see Figure 2.

Contributions. In summary, we significantly expand the applicability of package-query technology to handle very large problems with potentially tight constraints via the following contributions:

- A novel hierarchical strategy, called Progressive Shading, for finding high-quality package tuples that avoids the pitfalls of the SketchRefine approach (Section 2).
- An effective and efficient partitioning scheme, Dynamic Low Variance, for creating the hierarchical data partitions needed by Progressive Shading and handling outlying data well, together with an analytical comparison to kd-tree that verifies DLV's superior behavior (Section 3).
- A novel heuristic, Dual Reducer, for a very fast approximate solution of the final ILP encountered in Progressive Shading that uses a simple pruning strategy and a mechanism to guarantee solvability (Section 2.4), along with an optimized and highly parallelized LP solver, Parallel Dual Simplex, for accurate solution of LPs encountered in Progressive Shading (Section 2.3).
- A thorough experimental study showing that, unlike SketchRefine, Progressive Shading is scalable beyond hundreds of millions of tuples and, even for smaller relations, it can solve "hard" package queries for which SketchRefine suffers from false infeasibility. When both algorithms can produce feasible packages, Progressive Shading is faster and the solution packages have better objective values (Section 4.2). Notably, as part of this study, we define a novel *hardness* metric and provide a way to generate queries of a specific hardness, thereby providing a means and

a benchmark to systematically evaluate package-query solvers (Section 4.1).

2 PROGRESSIVE SHADING

The key challenge with directly solving the large ILPs that arise from package queries over large relations is that current solvers require that the corresponding LPs (where the integrality constraints are removed) fit into memory. Both SketchRefine and Progressive Shading algorithms avoid this problem by partitioning the large relation into smaller groups that fit in memory and then formulating small ILPs based on the representative tuples corresponding to these groups, thereby obtaining an approximate solution to the original ILP. These two algorithms, however, use very different strategies to obtain these small ILPs.

While SketchRefine "refines" the sketch solution by iteratively replacing each chosen representative tuple with the group's tuples, Progressive Shading first *augments* the sketch solution with additional "promising" representative tuples and then replaces all the chosen representative tuples with their group's tuples at once. In doing so, Progressive Shading tries to make each intermediate LP as large as possible by augmentation (via Neighbor Sampling); this improves the quality of the ILP solution of the original tuples relative to SketchRefine by not eliminating potentially high-quality tuples from consideration too early. More specifically, the algorithm tries to always solve an LP or ILP where the number of variables is close to, but does not exceed, an upper bound α. We call α the *augmenting size*; it is chosen so that an LP with α variables fits in memory and can be solved relatively fast, e.g., within 1 second for interactive performance [27].

Hierarchy of Relations. Progressive Shading relies on a hierarchy of relations of $L+1$ layers where layer 0 is the original relation and each layer $l \geq 1$ is the relation comprised of r_l representative tuples obtained after grouping the n_{l-1} tuples in layer $l-1$. That is, layer $l-1$ is partitioned into r_l groups with *downscale factor* $d_f = n_{l-1}/r_l$. So the *downscale factor* d_f is the average number of tuples per group.

Given d_f, the depth L of the hierarchy is the smallest number of layers such that the final layer L has a size at most α. That is, for a relation having n tuples and a *downscale factor* d_f, the final layer L has a size approximately $n/(d_f)^L \leq \alpha$, so that the minimal number of layers is $L = \lceil \log_{d_f}(n/\alpha) \rceil$. See Figure 3 for an example.

A group in layer $l \in [0..L]$ is defined by intervals $[a_j, b_j]$ where $-\infty \leq a_j < b_j \leq \infty$ for each attribute j such that all groups are non-overlapping. A tuple t belongs to the group if and only if $t.j \in [a_j, b_j]$ for all j, where $t.j$ is the attribute j of tuple t.

To compute the hierarchy of relations, we apply our partitioning algorithm, Dynamic Low Variance (Section 3.2), iteratively from layer 0 to layer $L-1$ with a *downscale factor* d_f. For ease of presentation, we largely ignore the effects of local predicates on the solvability and optimality of the solution package; see [22, Appendix E] for a brief discussion of how to mitigate the decreasing accuracy of representative tuples as local predicates select fewer tuples.

Progressive Shading overview. A high-level view of Progressive Shading is presented in Algorithm 1. Given the hierarchy of relations, along with the *augmenting size* α, Progressive Shading processes a package query by starting with the set S_L of all

Algorithm 1 Progressive Shading

Input: Q := package query
Parameter: α := augmenting size
1: $S_L \leftarrow$ set of indices of all representative tuples at layer L
2: $l \leftarrow L$
3: **while** $l > 0$ **do**
4: $\quad S_{l-1} \leftarrow$ Shading(l, α, S_l, Q)
5: $\quad l \leftarrow l - 1$
6: $S^* \leftarrow$ Dual Reducer (Q, S_0)
return S^*

Algorithm 2 Shading

Input: l := layer $l > 0$
α := *augmenting size*
S_l := set of indices of potential candidates at layer l
Q := package query
1: $P \leftarrow$ Formulate LP $(Q[S_l])$
2: $x^* \leftarrow$ Parallel Dual Simplex (P)
3: $S'_l \leftarrow \{i \in S_l | x^*_i > 0\}$ ▷ $S'_l \subseteq S_l$
4: $S_{l-1} \leftarrow$ Neighbor Sampling(l, α, S'_l)
return S_{l-1}

potential candidates—i.e., the set of all representative tuples—in layer L (line 1) and then iterates through the hierarchy down to layer 0 using Shading (Algorithm 2) to return a set S_{l-1} of at most α potential candidates from layer $l-1$ given the set of potential candidates S_l from layer l (line 4). At layer 0, Progressive Shading produces the final solution package from the package query $Q[S_0]$ using Dual Reducer (Section 2.4), our heuristic ILP solver specifically designed to be efficient when solving ILPs arising from package queries (line 6). We describe the various components of Progressive Shading in the following subsections.

2.1 Shading

Shading starts by formulating a package query $Q[S_l]$ from the tuples in S_l, which leads to an ILP. The algorithm then formulates an LP by removing the integrality conditions of the ILP (line 1).

It then solves the LP using Parallel Dual Simplex (Section 2.3) — our efficient LP solver specifically designed to exploit the fact that package queries have a very low number of constraints m (line 2). The LP solution x^* serves only to seed the initial set S'_l of potential candidates, i.e., S'_l comprises tuples with positive coefficients in x^* (line 3). The final step is to augment and expand the representative tuples in S'_l to S_{l-1} (line 4) via the Neighbor Sampling algorithm (Section 2.2).

A potential concern is that expanding the representative tuples in S'_l might generate an excessive number of candidate tuples at layer $l-1$; that is, the expected number of layer-$(l-1)$ tuples $d_f|S'_l|$ will exceed α and removal of tuples, rather than augmentation up to size α, will be required. This scenario is unlikely, though, because (1) d_f is typically small (Section 3.1), and (2) for package queries, the number $|S'_l|$ of positive coefficients in x^* is typically small in that $|S'_l| \leq \lceil m + \|x^*\|_1 \rceil \ll \alpha$ where m is the number of constraints and $\|\cdot\|_1$ is the L1 norm (see Section 2.4 for a proof). If this scenario

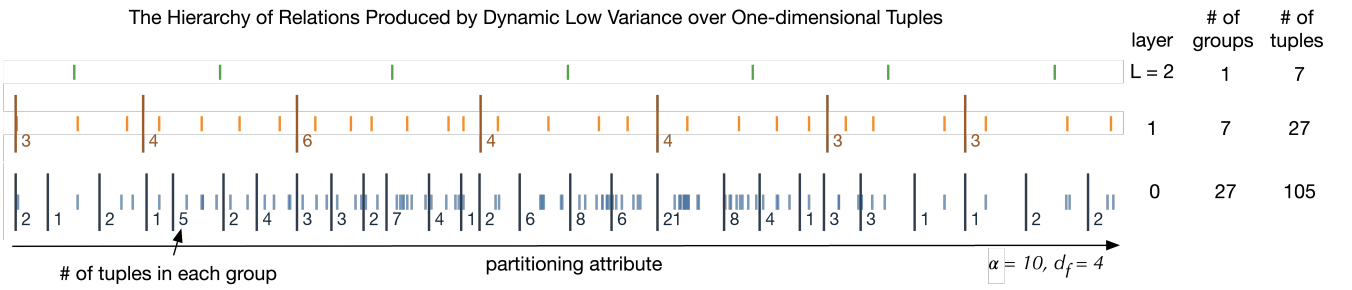

Figure 3: A 3-layer hierarchy of relations produced by Dynamic Low Variance with a *downscale factor* of 4. Each short colored bar represents a tuple in the hierarchy with long vertical black lines denoting partition boundaries.

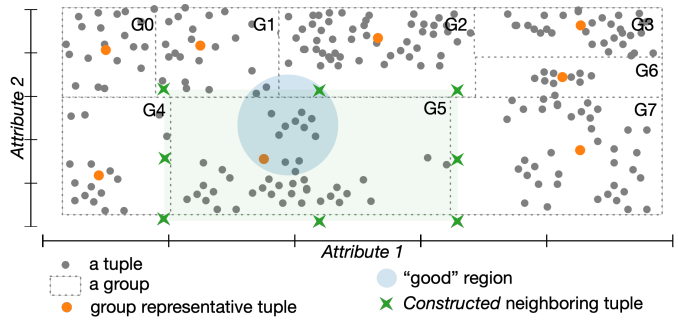

Figure 4: Groups G1, G2, G4, and G7 are neighboring to G5 since they contain the constructed neighboring tuples.

occurs, we can remove tuples in order of worst objective-value coefficient first until the number of layer-$(l-1)$ tuples is at most α.

Mini-Experiment 1. *Does replacing the LP solution with an ILP solution in Shading improve overall optimality?*
No. We observed no improvement in Progressive Shading's optimality or solvability when replacing the LP solution (line 2 of the Shading algorithm) with an ILP one. As LPs are faster to solve than ILPs, we prefer the LP formulation. See [22, Figure 13] for details.

2.2 Neighbor Sampling

Given the solution tuples S'_l at layer l, Neighbor Sampling in line 4 of Algorithm 2 selects tuples S_{l-1} from layer $l-1$. First, it replaces l-layer tuple g in S'_l with the $l-1$-layer tuples of the group g via GetTuples$(l-1, g)$ (line 2 of Algorithm 3). It then augments this set with tuples from neighboring groups.

Figure 4 shows a typical representation of 2D groups, demarcated by horizontal and vertical lines. In general, a group can have more than two attributes, with $[a_j, b_j]$ specifying the group boundaries along attribute j. Each group is represented by its average tuple (the orange dot in Figure 4). Suppose the blue circle represents a "good" region of tuples that are likely to be found in the optimal package. G5's representative tuple (orange dot) lies within this good region and is selected in the candidate solution set S'_l. If we only select G5's tuples for the next Shading iteration, we would miss out on tuples in G1 that lie within the good region. These are *hidden outliers* — tuples that are potentially in the final solution but are hidden as their groups' representative tuples are far from the "good" region. We want an algorithm that can add tuples from these neighboring groups which are identified by some measure of "closeness" to the selected group G5.

Algorithm 3 Neighbor Sampling

Input: $l :=$ layer $l > 0$
$\alpha :=$ augmenting size
$S'_l :=$ set of indices of tuples selected by the LP solution.

1: $\epsilon \leftarrow \min_{t.j \neq \tilde{t}.j} |t.j - \tilde{t}.j|$
2: $S_{l-1} \leftarrow \cup_{g \in S'_l}$ GetTuples$(l-1, g)$
3: $\bar{S}'_l \leftarrow \emptyset$ ▷ *The complement of S'_l*
4: **while** $|S'_l| > 0$ and $|S_{l-1}| < \alpha$ **do**
5: $g \leftarrow \arg\max_{g \in S'_l}$ ObjVal(g)
6: $S'_l \leftarrow S'_l \setminus \{g\}$ ▷ *S'_l is a max-priority queue*
7: $\bar{S}'_l \leftarrow \bar{S}'_l \cup \{g\}$
8: $A_g \leftarrow \{[a_j, b_j], j = 1, ..., k\}$
9: $T \leftarrow \{a_1 - \epsilon, \frac{a_1+b_1}{2}, b_1 + \epsilon\} \times ... \times \{a_k - \epsilon, \frac{a_k+b_k}{2}, b_k + \epsilon\}$
10: **for** each $t \in T$ **do**
11: $g' \leftarrow$ GetGroup(l, t)
12: **if** $g' \notin S'_l \cup \bar{S}'_l$ **then** ▷ *If we have not seen g' before*
13: $S'_l \leftarrow S'_l \cup \{g'\}$
14: $S_{l-1} \leftarrow S_{l-1} \cup$ GetTuples$(l-1, g')$
return $S_{l-1}[:\alpha]$ ▷ *Return α highest objective tuples.*

Without loss of generality, we present Neighbor Sampling (Algorithm 3) assuming an objective maximization query. One can replace 'max/highest' with 'min/lowest' for objective minimization queries. We select a group g with the highest objective value ObjVal(g) (line 5). Group g is defined by a set of intervals $A_g = \{[a_j, b_j], j = 1, ..., k\}$ (line 8). We *construct* a neighboring tuple t that lies "just outside" group g by setting each attribute $t.j$ equal to $a_j - \epsilon$, $b_j + \epsilon$, or $(a_j + b_j)/2$ where ϵ is the smallest positive distance between any two tuples in layer l over some attribute (line 1). We let T be the set of all such tuples (line 9); note that $|T| = 3^k$, where k is the number of attributes. We now find the group g' (GetGroup(l, t), line 11), which contains the constructed tuple t, and add its representative tuple to S'_l and all its constituent tuples to S_{l-1} (line 13,14). The efficiency of Neighbor Sampling critically relies on the efficiency of GetGroup(l, t). A naive implementation

of GetGroup(l, t) would be to linearly scan all the groups to find where tuple t belongs. We show in [22, Appendix D.2] that Dynamic Low Variance can achieve sub-linear time complexity for the function GetGroup(l, t). This sampling of neighboring tuples continues as long as $|S_{l-1}| < \alpha$ (line 4).

Mini-Experiment 2. *Does replacing Neighbor Sampling with a random sampling of representative tuples impact the overall performance of Progressive Shading?*
We ran query Q1 SDSS described in Table 1 with query-hardness levels $\tilde{h} \in \{1, 3, 5, 7, 9, 11, 13\}$ (Section 4.1). For each $\tilde{h}$, we randomly sampled 5 sub-relations of size 10 million representing 5 queries for a total of 35 queries. We compared the results between two Progressive Shading variants: one with Neighbor Sampling and one where Neighbor Sampling is replaced by a random sampling of tuples. Progressive Shading with Neighbor Sampling solved all of the 35 package queries while the random-sampling variant solved one less. On average, the solvable queries showed a 7.67x improvement in the objective value when using Neighbor Sampling. Experiments with other queries yield similar results; see [22, Figure 15] for details.

2.3 Parallel Dual Simplex

A key ingredient in Progressive Shading is the LP solver. Typical LP sizes range from hundreds of thousands to tens of millions of variables. The standard dual simplex algorithms in commercial systems such as Gurobi or CPLEX are sequential and make no assumptions on the number of variables versus the number of constraints. In this generic setting, prior works [4, 15] have tried to efficiently parallelize dual simplex for up to 8 processing cores. Specifically, in [15], the authors observed a 2.34x speedup at 8 cores, with 65% of the execution effectively parallelized.

We introduce a novel algorithm, Parallel Dual Simplex, that achieves superior speedup by exploiting the special structure of the ILPs that arise when solving package queries. In textbook ILPs (such as set cover [32], unit commitment [19], knapsack sharing [11], and traveling salesman [23]), the number m of constraints is a polynomial function of the number n of variables. In contrast, a package query ILP has a constant number of constraints m that is much smaller than n. By exploiting this structural difference, we greatly simplify our dual simplex implementation and are also able to efficiently parallelize most of the dual simplex sub-procedures. Roughly speaking, in dual simplex, we quickly move from one solution to another better one by selecting a good direction via *pivoting* [28]. Moving between solutions involves multiplications of an $n \times m$ matrix by an m-vector, which can be parallelized over n. Furthermore, the search for a good direction is a sequential operation but can be parallelized efficiently as we observed in our experiments assuming that we have a few constraints m and a huge number of variables n. See [22, Appendices B and C] for technical details.

Mini-Experiment 3. *How well does Parallel Dual Simplex scale with more cores?*
We found that our Parallel Dual Simplex algorithm can scale up to at least 80 cores, attaining a 4.79x speedup, with 80% of the execution effectively parallelized—a significant improvement over generic parallel dual simplex implementations. See [22, Figure 12] for details.

Algorithm 4 Dual Reducer

Input: Q := package query
S := set of indices of n tuples in the relation
Parameter: q := initial size of the sub-ILP

1: $P \leftarrow$ Formulate LP$(Q[S])$
2: $x^* \leftarrow$Parallel Dual Simplex (P)
3: $E \leftarrow \sum_{i=1}^{n} x_i^*$
4: $P' \leftarrow P$ where the upper bound of each variable is E/q
5: $y^* \leftarrow$Parallel Dual Simplex (P')
6: $S' \leftarrow \{i | x_i^* > 0 \vee y_i^* > 0\}$
7: $P^* \leftarrow$ Formulate ILP$(Q[S'])$
8: $S^* \leftarrow$ ILPSolver(P^*)
9: **while** $S^* = \emptyset$ and $q < n$ **do** ▷ *Fallback mechanism*
10: $\quad q \leftarrow \min(2q, n)$
11: $\quad$ Uniformly sample $S_u \subseteq \{i | i \notin S'\}$ such that $|S_u| = q - |S'|$
12: $\quad S' \leftarrow S' \cup S_u$
13: $\quad P^* \leftarrow$ Formulate ILP$(Q[S'])$
14: $\quad S^* \leftarrow$ ILPSolver(P^*)
15: **return** S^*

2.4 Dual Reducer

Dual Reducer is a novel heuristic (Algorithm 4) for efficiently and approximately solving the final ILP encountered in Progressive Shading (line 7 of Algorithm 1). It is a type of Relaxation Enforced Neighborhood Search (RENS) heuristic [3, 9]. RENS are characterized by constructing a sub-ILP (to be solved by a black-box ILP solver) where most of the zero decision variables in the LP relaxation x^* are hard-fixed to 0.

Number of positive coefficients in the LP solution. Dual Reducer initially computes the LP solution x^* (line 1-2). For x^*, the theory of the simplex method [14] asserts that the number of *basic* variables that can take fractional values is at most the number of constraints m. Assuming that the upper bound of each variable is 1, the number of *non-basic* variables, which can either be 0 or 1, is therefore $n - m$, where n is the number of variables. Letting $E = \sum_{i=1}^{n} x_i^*$ (line 3) be the sum of all decision variables of x^*, i.e. the L1 norm of x^*, we see that the number of variables that are 0 is at least $\lfloor n - m - E \rfloor$. Note that most of the decision variables are 0 in x^* since $n \gg m + E$ and only a few variables are positive, i.e., at most $\lceil m + E \rceil$ of them. We can now use x^* to construct a reduced-size sub-ILP from the positive variables. Let q be the size of this sub-ILP.

Configuring q. If $q \approx E$, i.e. we have pruned out all zero-valued decision variables, we may end up with false infeasibility (the sub-ILP is infeasible but the ILP itself is feasible) or sub-optimality. If q is too large, then we may incur unnecessary and significant computational costs. The right value of q should be small enough to allow the sub-ILP to be solved within interactive performance by an off-the-shelf black-box ILP solver, (i.e. sub-second time), yet large enough to comfortably contain the typical solution sizes for package queries. E.g., package queries in our benchmark (Section 4.1) typically have solutions with 10 to 1000 tuples ($E \approx [10 -$

1000]), and setting $q = 500$ achieves the right balance of interactive performance and feasibility. See [22, Mini-Experiment 7] for the impact of q on the performance of Dual Reducer.

Sub-ILP. From x^* and q, Dual Reducer constructs an auxiliary LP P' such that its solution has approximately q positive variables (lines 4-5). We observe that the sum of all decision variables of x^* is E when the upper bound of each variable is 1. Hence, by limiting the upper bound of each variable to E/q, we hope to have at least q positive variables. This simple modification effectively forces the LP solver to distribute its choices evenly across the tuples to produce q positive decision variables in y^*. Dual Reducer now formulates and solves the sub-ILP using tuples with positive coefficients in both the initial and the auxiliary LP solutions, x^* and y^* (lines 6-8).

Fallback mechanism. Unlike other RENS heuristics, Dual Reducer has a graceful fallback mechanism to handle false infeasibility if q is insufficiently large (line 9). Dual Reducer doubles q and randomly samples more tuples to include in the sub-ILP from the original relation (lines 11-12) until it includes the full relation. In practice, we observed that many of the difficult queries could be solved after one or two fallback iterations, i.e., doubling or quadrupling the initial sub-ILP size, without falling all the way back to the original relation.

Mini-Experiment 4. *Does replacing the Auxilary LP with a random sampling of tuples from S to formulate a sub-ILP of size q impact the overall performance of Dual Reducer?*
We ran query Q1 SDSS described in Table 1 with query-hardness levels $\tilde{h} \in \{1, 3, 5, 7, 9, 11, 13\}$ (Section 4.1). For each $\tilde{h}$, we randomly sampled 5 sub-relations of size 1 million representing 5 queries for a total of 35 queries. We compared the results between two Dual Reducer variants: one with the Auxiliary LP P' and one with a random sampling, i.e., replacing line 6 of Algorithm 4 with $S' \leftarrow \{i | x^*_i > 0 \vee u_i < q/n\}$ where $u_i \sim \mathcal{U}(0, 1)$. Dual Reducer with Auxiliary LP solved all 35 queries, while Dual Reducer with random sampling solved only 25. For queries solved by both variants, we observed an improvement in the objective value by 1.135x on average when using Dual Reducer with Auxiliary LP. Results for other queries were similar; see [22, Figure 16] for details.

3 PARTITIONING ALGORITHM

Dynamic Low Variance (DLV) is a novel partitioning algorithm that works with multidimensional tuples (Section 3.2) and very large relations ([22, Appendix D.2]). The algorithm relies on the 1-D Dynamic Low Variance (1-D DLV) subroutine that iteratively selects and partitions a relation one attribute at a time. 1-D DLV is unlike traditional partitioning algorithms such as kd-tree: in one iteration, it partitions an attribute using $p \geq 2$ flexible intervals instead of just two intervals separated by the attribute's mean.

Definition 1 (p-Partition). *Given a set S of tuples, and a vector $d = (d_0, d_1, \ldots, d_p)$ where $p \geq 1$ and $-\infty = d_0 < d_1 < \ldots < d_{p-1} < d_p = \infty$, the p-partition $\mathcal{P}_d(S, j)$ of the set S over an attribute j is the disjoint partition $\{P_1, P_2, \ldots, P_p\}$ of S such that $P_i = \{t \in S : d_{i-1} \leq t.j < d_i\}$ for $1 \leq i \leq p$, where $t.j$ is the attribute j of tuple t.*

Algorithm 5 1-D Dynamic Low Variance

Input: β := bounding variance
S := set of k-dimensional tuples of size n
j := attribute to partition

```
1:  S̃ ← list of tuples in S sorted in increasing order of attribute j
2:  V ← ∅
3:  d ← {−∞, ∞}
4:  for each t ∈ S̃ do
5:      if σ²(V ∪ {t.j}) > β then        ▷ σ² is the variance function
6:          d ← d ∪ {t.j}
7:          V ← ∅
8:      V ← V ∪ {t.j}
9:  d̃ ← vector of values in d sorted in increasing order
10: return P_d̃(S, j)
```

3.1 1-D Dynamic Low Variance

The core idea of 1-D DLV is to minimize the variance of each subset P_i in a p-partition by dynamically allocating more P_i's to partition a spread-out set of attribute values and fewer P_i's to partition a concentrated one. The procedure is given as Algorithm 5.

Given a specified value $\beta > 0$ called the *bounding variance*, 1-D DLV iterates through the attribute values in increasing order (line 1). The algorithm keeps track of a running variance of the values grouped so far (line 9). Once this variance exceeds β (line 5), it places a delimiter between the current tuple and the previous one and resets the running variance (lines 6-7).

Configuring d_f. Recall that a smaller *downscale factor* d_f in Section 2 yields a smaller expected number of tuples in by each group. A representative tuple more accurately represents its group's tuples if the group has fewer, more concentrated tuples. We also augment the solution package S'_l at every Shading iteration with neighboring representative tuples (Section 2.2). If we have smaller, and hence more, neighboring groups, we can add more representative tuples to S'_l during Neighbor Sampling up to the *augmenting size* α and thus better capture hidden outliers. However, the smaller the d_f, the higher the computational cost of Progressive Shading as the depth of the hierarchy of relations increases. We observed that $d_f \approx [10 - 1000]$ achieves the right balance between accuracy and computation cost.

Configuring β. In Progressive Shading, given a *downscale factor* d_f, one wishes to find a bounding variance β such that the p-partition produced by 1-D DLV has $p \approx n/d_f$ where n is the relation size. However, 1-D DLV with a single bounding variance β can fail to achieve certain small target values for d_f, especially when the variance of the distribution is low; see Figure 5. This is an issue since Progressive Shading requires d_f to be very small ($d_f \approx [10 - 1000]$). DLV overcomes this issue by using multiple bounding variances on multiple attributes and as a result extends to multidimensional settings.

3.2 Dynamic Low Variance

DLV is displayed as Algorithm 6. It is a divisive hierarchical clustering algorithm [29] where all tuples start in one cluster (line 2) and splits are performed recursively until we reach $\approx |S|/d_f$ clusters

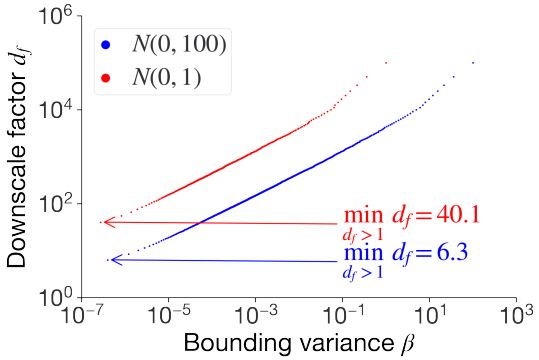

Figure 5: The observed *downscale factor* d_f for different bounding variances β under two normal distributions $\mathcal{N}(0, 1)$ and $\mathcal{N}(0, 100)$.

Algorithm 6 Dynamic Low Variance

Input: $S :=$ set of k-dimensional tuples
$d_f :=$ *downscale factor*

1: $(c_1, c_2, \ldots, c_k) \leftarrow$ GetScaleFactors(S, d_f).
2: $\mathcal{P} \leftarrow \{S\}$ ▷ *$\mathcal{P}$ is a max-priority queue*
3: **while** $|\mathcal{P}| < |S|/d_f$ **do**
4: $\quad P^* \leftarrow \arg\max_{P \in \mathcal{P}} |P| \max_j \sigma^2(P, j)$
5: $\quad j^* \leftarrow \arg\max_j \sigma^2(P^*, j)$
6: $\quad \beta \leftarrow c_{j^*} \sigma^2(P^*, j^*)/{d_f}^2$
7: $\quad \mathcal{P} \leftarrow (\mathcal{P} \setminus \{P^*\}) \cup$ 1-D Dynamic Low Variance(β, P^*, j^*)
return $\mathcal{P}$

(line 3) where $|S|$ is the number of tuples. The splitting always prioritizes the cluster P^* with the maximum highest total variance using a max priority queue (line 4) where $\sigma^2(P, j)$ is the variance of the attribute j of tuples in P. For cluster P^*, we partition on the attribute j having the highest variance (line 5). As discussed below, the bounding variance β is set in such a way that the p-partition for the cluster P^* produced by 1-D DLV has approximately d_f subsets (lines 6-7).

Intuitively, DLV is analogous to iteratively partitioning the terrain squares in order of highest squares first in our stylized example (Figure 1) where each iteration corresponds to partitioning a square into $d_f = 4$ smaller squares. Hence, the first heuristic is to come up with a bounding variance β (line 6) so that in each iteration, 1-D DLV partitions P^* into approximately d_f subsets (line 7). Let σ^2 be the variance of the partitioning attribute of P^*. We observed that the appropriate form for β is $c\sigma^2/{d_f}^2$ for a constant $c > 0$ since P^* is partitioned into approximately d_f subsets so the variance of each subset is expected to decrease by a factor of ${d_f}^2$. Moreover, the value of c depends on the distribution of P^* and we can accurately find such c, by simply binary searching β for each P^* assuming that the number of partitioning subsets of P^* is a decreasing function of β. However, this approach requires us to do multiple executions of 1-D DLV over P^* in each iteration and hence is slow in practice. To avoid this, we can approximate c_j for each attribute j before the iterations via the GetScaleFactors function (line 1) which essentially samples the attribute values and then does a binary search. See [22, Appendix D.1] for the details of the function. For our datasets, we found $c = 13.5$ to work well.

The second heuristic is to choose a ranking that best captures the variability of a multi-dimensional subset $P \in \mathcal{P}$ (line 4). For each subset of tuples, one can compute the variance or the total variance (i.e., variance times set size) for each attribute and take the maximum over all the attributes. We observed empirically that using the total variance would produce much better solutions compared to using the variance. There are several advantages to using DLV:

- It partitions on multiple attributes and produces partitions for any given number of partitioning subsets.
- The actual DLV partitioning operation is usually executed on a partition much smaller than S. This allows sorting algorithms on these smaller subsets to be much faster and cache-friendly.
- The average number of passes through the relation is $O(\log_{d_f} n/d_f)$ where n is the relation size and d_f is the *downscale factor*.

In [22, Appendix D.2], we show how to extend DLV to run on large relations via a bucketing scheme.

3.3 Comparison to KD-TREE

Partitioning score. Partitioning of a relation groups similar tuples together. A representative tuple can then be computed as an average over the similar tuples in the group. This similarity can be quantified by the tuples' distances to the representative. If we take the average of the squared distances between these tuples and the representative then this measure corresponds to the variance of the tuples' attributes. Therefore, the variance of the tuples' attributes reflects, on average, how spread out or clustered the group's tuple attribute values are and thus the similarity of tuples within the group. A good partitioning algorithm will create groups with more tightly clustered tuples, i.e., low within-group variance. Consequently, a useful measure of how well a partitioning algorithm performs will reflect the changes in within-group variance before and after partitioning. We define the *Ratio Score* as such a measure. For simplicity, we will restrict our analysis to partitions over one-dimensional tuples:

Definition 2 (Ratio score). *For the p-partition $\mathcal{P}_d(S)$ of the set S of one-dimensional tuples, let σ_i^2 be the variance of the tuple values in partition P_i $(1 \le i \le p)$ and $\sigma^2 > 0$ be the variance of tuple values in the unpartitioned set S. The ratio score $z(\mathcal{P}_d(S))$ is $\sum_{i=1}^{p} \sigma_i^2/\sigma^2$.*

Intuitively, the ratio score is the ratio between the sum of the subsets' variance and the set's variance. Hence, the lower the score, the better the partitioning algorithm. The lowest score is 0 when all the subsets have a variance of 0. On the other hand, if all of the subsets are empty except one $P_{i'}$ then $\sigma_i^2 = 0$ for $i \neq i'$ while $\sigma_{i'}^2 = \sigma^2$. Hence, the score is 1 for such a trivial partition. Ratio scores that exceed 1 are possible, as shown in Theorem 1 below.

KD-TREE versus 1-D DLV. KD-TREE is also a divisive hierarchical clustering algorithm [29] where a cluster is always split into two smaller clusters using the mean value. For generating up to several thousands of clusters, KD-TREE is more efficient than other traditional clustering algorithms since each pass through the relation essentially doubles the number of clusters produced. Once the number of generated clusters goes beyond millions, however,

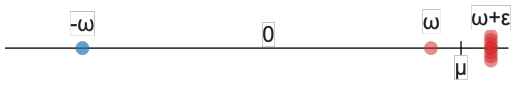

Figure 6: For a distribution consisting of one value each for $-\omega$ and ω and many values at $\omega + \epsilon$, the first KD-TREE split is at μ between ω and $\omega + \epsilon$, forcing a grouping of the highly discrepant values $-\omega$ and ω.

its performance deteriorates. In Brucato et al. [5], a cluster P_i is considered for splitting if it satisfies one of the two conditions: (1) its size $|P_i|$ is more than *size threshold* $\tau \geq 1$; and (2) its radius r_i is more than *radius limit* $\omega \geq 0$ where $r_i = \max_{x \in P_i} |x - \mu(P_i)|$ and $\mu(P_i)$ is the mean of P_i. The following result shows that on some data sets the clustering performance of KD-TREE degrades totally while 1-D DLV attains almost perfect clustering.

THEOREM 1. *For any radius limit $\omega > 0$, there exists a sequence $\{S_n\}$ of sets of one-dimensional tuples whose variances converge to 0 such that for any size threshold $\tau \geq 2$, KD-TREE's ratio score tends to ∞ as $n \to \infty$. On the other hand, using 1-D DLV with a bounding variance $\beta = 24\sigma^2(S_n)/|S_n|^2$, the ratio score converges to 0.*

We include the full proofs of our theoretical results in [22, Appendix A]. To prove Theorem 1, we construct a sequence $\{S_n\}$ of sets of one-dimensional tuples as in Figure 6. We force KD-TREE to group two very dissimilar values by exploiting the fact that the splitting intervals of KD-TREE are fixed as long as the mean of the values does not change. 1-D DLV, on the other hand, with an appropriate choice of bounding variance, overcomes this issue. Indeed, we now show that 1-D DLV has a low ratio score for virtually any large relation. Specifically, for any set of n one-dimensional tuples, 1-D DLV, using the above bounding variance, achieves an $O(1/n)$ ratio score. Moreover, the corresponding partitioning is nontrivial in that there exist partitions with at least two tuples, i.e., $p < n$.

THEOREM 2 (UNIVERSAL BOUNDED RATIO SCORE). *Let S be a set of one-dimensional tuples of size $n \geq 2$ with variance $\sigma^2 > 0$. Then 1-D DLV with a bounding variance $\beta = 24\sigma^2/n^2$ will produce a p-partition $\mathcal{P}_d(S)$ where $p \leq (3/4)n + 1/2$—so that the partitioning is nontrivial—and $z(\mathcal{P}_d(S)) \leq 24/n$.*

At a high level, our proof proceeds by estimating the number of so-called *critical intervals*, which are intervals of consecutive values in increasing order such that the two endpoints of any such intervals cannot be in one partitioning subset as it would violate the above bounding variance β. This, in turn, allows us to upper-bound the number of partitioning subsets containing a single value, i.e., not all partitioning subsets will contain a single value and thus upper-bound p as well. As a result, the ratio score is bounded by $24/n$ where n is the number of tuples.

DLV in practice. Figure 7 shows the ratio score z of various algorithms using the same *downscale factor* d_f partitioning on a normal distribution $\mathcal{N}(0, 1)$ with 10^5 samples.

> **Mini-Experiment 5.** *How efficient is DLV compared to KD-TREE when producing a large number of groups?*
> We ran DLV and the KD-TREE implementation as in [5] to partition the dataset TPC-H described in Section 4.1. DLV partitioned a relation of 10^8 tuples in 138s using 80 cores to produce approximately 10^6 groups while KD-TREE executed in 300s to produce approximately 10^3 groups. (KD-TREE is not well-suited to produce as many groups as DLV due to efficiency issues as well as the inability to directly control the number of groups produced). For a relation of 10^9 tuples, DLV took 1827s to produce approximately 10^7 groups while KD-TREE ran out of memory.

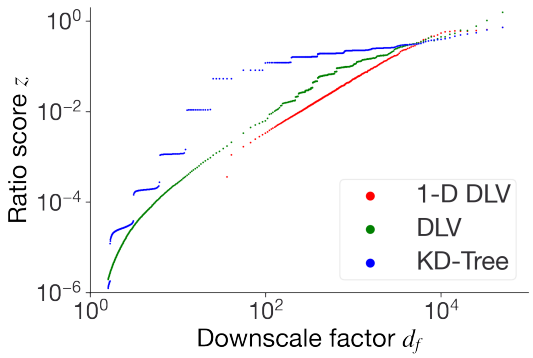

Figure 7: DLV outperforms KD-TREE and performs as well as 1-D DLV for various values of *downscale factor*.

4 EVALUATION

In this section, we demonstrate experimentally that PROGRESSIVE SHADING is very effective at overcoming the false infeasibility issues of the prior art (i.e., failing to derive a solution for feasible queries) while achieving superior scalability. We first describe the setup of our evaluation, including datasets, queries, and metrics, and then proceed to showcase our results.

4.1 Experimental Setup

Software and platform. We use POSTGRESQL v14.7 for our experiments to use built-in features such as range types to store DLV's partitioning information and GiST indexes over these range types. The main algorithms are implemented in C++17, which uses the *libpq* library as an efficient API to communicate with POSTGRESQL and the *eigen* library for efficient vector/matrix operations. For parallel implementation, we use C++ OpenMP for multi-processing computation [7]. For solving a sub-ILP in DUAL REDUCER, we use GUROBI v9.5.2 as our black-box ILP solver [10]. We run all experiments on a server with Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz with 377GB of RAM and 80 physical cores, running on Ubuntu 20.04.4 LTS. Our implementation is available at [22].

Datasets. We demonstrate the performance of our algorithms using both real-world and benchmark data. The real-world dataset consists of 180 million tuples extracted from infrared spectra (APOGEE/APOGEE-2) of the Sloan Digital Sky Survey (SDSS) [1]. For the benchmark dataset, we use the LINEITEM table from TPC-H V3 [31] with a scale factor of 300; the table contains 1.8 billion tuples. In order to make results comparable across the two datasets, we use the same query structure with constraint bounds that are set to achieve a specific query hardness level given the mean and standard deviation of the attributes in the dataset.

Queries. Given a dataset, we have developed a novel method for systematically generating queries of varying hardness, rather than generating queries in an ad hoc manner. This approach allows comprehensive benchmarking and yields a better empirical assessment of the generalizability of a technique to other data sets or package queries, which can be arbitrarily easy or hard. Specifically, we use a package query template and systematically vary the constraint bounds to expand or shrink its feasibility region. As a simple example, a package query with the constraint $\sum_j x_j < b$ is trivially feasible for $b = \infty$ and infeasible for $b < 1$ (since the x_i's are integer). The complexity of finding a solution also depends on the objective function and the shape of the feasible region.

Query Hardness. To precisely define query hardness, let $\mathcal{E}$ be the expected *package size*, i.e., the expected number of tuples in the solution to a package query. Without loss of generality, consider a constraint, C_i of the form $\sum_j a_{ij}x_j < b_i$. Suppose attribute A_i is a random variable with mean μ and variance σ^2. Then with a large enough $\mathcal{E}$ and by the central limit theorem, $\mathcal{E}^{-1}\sum_{j=1}^{\mathcal{E}} A_i$ follows a normal distribution $\mathcal{N}(\mu, \sigma^2\mathcal{E}^{-1})$. Constraint C_i can be reformulated as $\mathcal{E}^{-1}\sum_{j=1}^{\mathcal{E}} A_i < b_i/\mathcal{E}$. The probability, $P(C_i)$, that a random sample of $\mathcal{E}$ tuples satisfy C_i is simply given by the cumulative distribution function (CDF) of the normal distribution $\mathcal{N}(\mu, \sigma^2\mathcal{E}^{-1})$ evaluated at $b_i/\mathcal{E}$. With m constraints, $C_1, ..., C_m$, and assuming the attributes are independent for the sake of simplicity, the probability that a random sample of $\mathcal{E}$ tuples satisfy all the constraints is $P(C_1, C_2, ...C_m) = \prod_i^m P(C_i)$. Since the chances of satisfying a harder query's constraints with a random sample of tuples are much lower, we can define *hardness* as follows: $\tilde{h} := -\log_{10}\prod_i^m P(C_i)$. Given a template package query with constraints $C_1, \ldots, C_m$, their bounds $b_1, \ldots, b_m$ as parameters, and the expected package size, $\mathcal{E}$, we can now instantiate a specific query of a specified hardness by setting the bounds accordingly. In particular, we can set $P(C_1) = P(C_2) = \cdots = P(C_m) = 10^{-\tilde{h}/m}$ and invert the CDF function to derive the bound b_i for which $P(C_i) = 10^{-\tilde{h}/m}$.

Table 1 provides information on the underlying data distribution statistics of both data sets, the package query templates, and the bounds set for query instances of a particular hardness level $\tilde{h}$, where $\tilde{h} \in \{1, 3, 5, 7\}$.

Approaches. Our evaluation contrasts three approaches:

- Gurobi ILP solver [10]: This is a state-of-the-art solver that computes solutions to the ILP problem directly, without any considerations of partitioning. It provides the gold standard with respect to accuracy but struggles to scale to large data sizes.
- SketchRefine [5]: The prior state-of-the-art in package query evaluation employs a data partitioning and divide-and-conquer strategy to achieve scalability.
- Progressive Shading: Our approach employs a multi-layer partitioning strategy that smartly *augments* the size of ILP subproblems to avoid false infeasibility, and a novel mechanism to parallelize and reduce solving time.

Metrics. We evaluate the efficiency and effectiveness of all methods. The first metric is *running time*. We measure the wall-clock time to generate a solution for each method. This includes the time taken to read data from PostgreSQL and the time taken for the method to produce the solution. In particular, for Progressive Shading and SketchRefine, the running time is computed when running 80 cores in parallel. (We use the parallel version of SketchRefine described in [5].) For Gurobi, only sequential execution using 4 cores is available. Therefore, we also include the running time of Progressive Shading using 4 cores in parallel. We limit the maximum running time of any method to 30 minutes. If a method fails to produce a solution within this time limit, it is registered as a failed run, i.e., no solution found.

The second metric is the *integrality gap*. Recall that the solution of the LP relaxations of an ILP is readily available because we can efficiently solve the LP problem using the Simplex algorithm [26]. Hence, we use the LP objective value as the upper bound for an ILP solution in a maximization problem, and as the lower bound in a minimization problem. The *integrality gap* for maximization is then defined as the ratio ILP objective over LP objective: $(Obj_{ILP} + \epsilon)/(Obj_{LP} + \epsilon)$ where $\epsilon = 0.1$ is required to avoid numerical instability when $|Obj_{LP}|$ is too small. For minimization, we simply invert the ratio. Therefore, it is always the case that the integrality gap is at least 1 assuming the objective is always positive.

Hyperparameters. We set hyperparameters as follows.

- *Gurobi's MIP gap*: We keep the default value of 0.1%. Gurobi will terminate when the gap of the lower and upper bound of the optimal objective value is less than 0.1% of the incumbent objective value.
- *SketchRefine's partitioning size threshold*: We find that the default setting proposed by SketchRefine [5] (10% of the relation size, or $\approx$ 10 partitions) results in infeasibility in the sketch phase for all queries with hardness $\tilde{h} > 2$ in our benchmark. We instead set the threshold to 0.1%. This increases the number of partitions ($\approx$ 1000) allowing for smaller groups with more similar tuples and better representatives leading to a higher solve rate, without degrading the performance of the kd-tree index.
- *Progressive Shading's augmenting size α and downscale factor d_f*: Using grid search, we find α = 100,000 and d_f = 100 to be optimal. Lower d_f would cause the partitioning time to be much greater (almost 3x longer) while higher d_f would cause the partitioning groups to be less accurate since each group would contain more tuples. Higher α significantly increases the query time with marginal gains to solution quality. Lower α results in a significant drop in optimality (3x worse). Case-by-case parameter tuning can be used, if needed, incurring a tuning overhead of up to 2 hours. However, the results in Section 4.2 indicated that the foregoing hyperparameter configuration works well with various query structures and hardnesses. See [22, Mini-Experiment 6] for details of the grid search.

4.2 Results

Query performance as relation size increases. Figure 8 demonstrates the performance of each method as the relation size increases, as well as the effect of increasing hardness on the running time and the integrality gap. The set of hardness values that we experimented with encompasses easy to average difficulty ($\tilde{h} \in \{1, 3, 5, 7\}$). We generate ten relation instances for each relation size by sampling independent sub-relations from the original dataset.

Table 1: Experimental Benchmark Q1 SDSS and Q2 TPC-H: The package query templates, underlying data statistics, and constraint bounds at different query-hardness ($\tilde{h}$) levels.

Q1 SDSS

```
SELECT PACKAGE(*) AS P FROM sdss R REPEAT 0
SUCH THAT 15 ≤ COUNT(P.*) ≤ 45 AND
SUM(P.j) ≥ b1 AND SUM(P.h) ≤ b2 AND
SUM(P.k) BETWEEN b3 AND b4
MINIMIZE SUM(P.tmass_prox)
```

Attribute	μ	σ
tmass_prox	14.45	14.96
j	14.82	1.562
h	14.05	1.657
k	13.73	1.727

$\tilde{h}$:	1	3	5	7
b_1	445.37	455.56	461.91	466.86
b_2	420.68	409.87	403.14	397.89
b_3	406.04	410.71	411.64	411.84
b_4	417.76	413.09	412.16	411.96

Q2 TPC-H

```
SELECT PACKAGE(*) AS P FROM tpch R REPEAT 0
SUCH THAT 15 ≤ COUNT(P.*) ≤ 45 AND
SUM(P.quantity) ≥ b1 AND SUM(P.discount) ≤ b2 AND
SUM(P.tax) BETWEEN b3 AND b4
MAXIMIZE SUM(P.price)
```

Attribute	μ	σ
price	38240	23290
quantity	25.50	14.43
discount	1912	1833
tax	1530	1485

$\tilde{h}$:	1	3	5	7
b_1	772.11	866.29	924.88	970.61
b_2	56456.81	44493.54	37051.09	31242.12
b_3	40864.32	44877.91	45680.35	45852.68
b_4	50935.68	46922.09	46119.65	45947.32

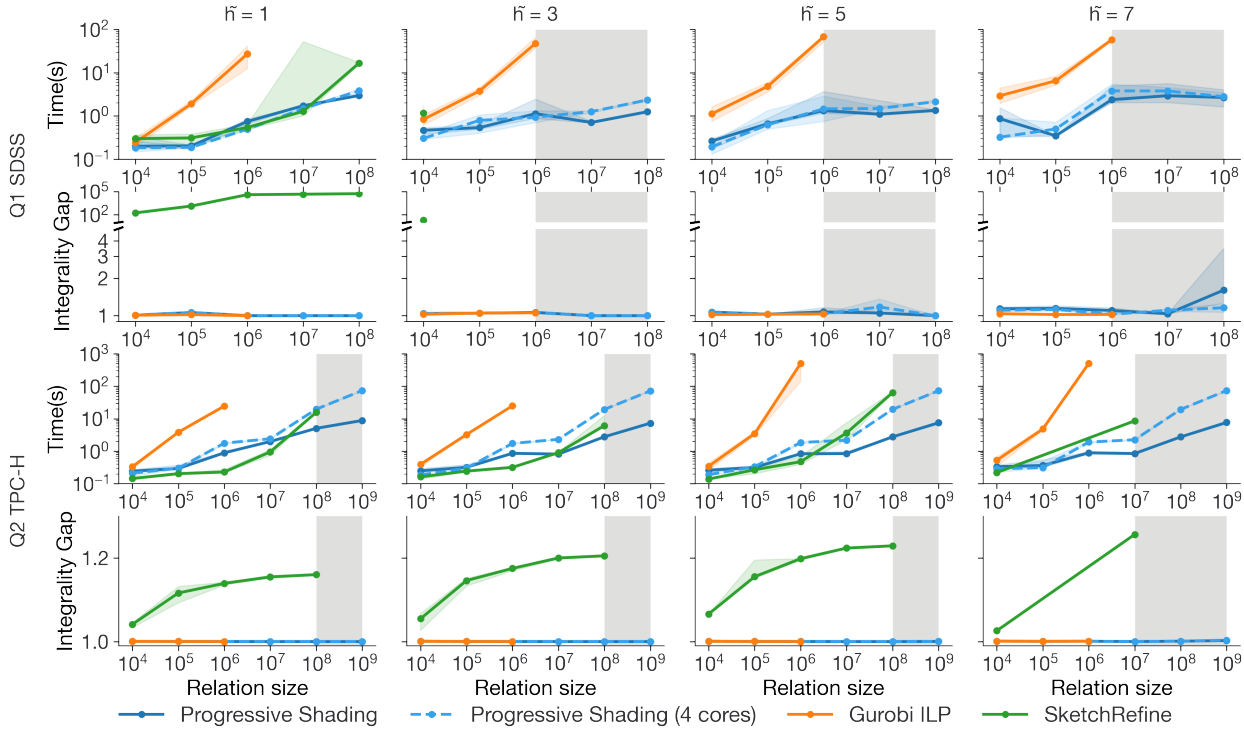

Figure 8: Query performance as relation size increases for Q1 SDSS and Q2 TPC-H. The point values represent the median of 10 runs and the error bands are the interquartile range (IQR) of the 10 runs.

In both queries, Gurobi only scales up to a size of one million tuples, and the running time grows exponentially with the relation size. SketchRefine scales relatively well up to ten million tuples, but cannot sustain scaling beyond that, since the size of refined queries increases linearly with the relation size. In addition, SketchRefine fails to find solutions for $\tilde{h} \geq 3$ for Q1 SDSS and $\tilde{h} = 7$ for Q2 TPC-H. In contrast, Progressive Shading always finds solutions in both queries and achieves well below 5s running time even for one billion tuples.

In terms of the integrality gap, Progressive Shading achieves close-to-optimal solutions for Q1 SDSS and Q2 TPC-H as seen by its integrality gap curve staying as close to that of Gurobi. SketchRefine, on the other hand, produces solutions with 20% worse objective in Q2 TPC-H. For Q1 SDSS, the extremely high value of integrality gap for SketchRefine is due to the fact that the objective column *tmass_prox* in SDSS has many zero values. This produces an LP solution with an objective value of 0. If SketchRefine only finds an ILP solution with a positive objective value, this value will be divided by $\epsilon = 0.1$, i.e., will be scaled by a factor of 10.

False infeasibility as hardness increases. We next examine the occurrence of false infeasibility in Progressive Shading and SketchRefine as the hardness level becomes very high, shrinking the feasible region ($\tilde{h}$ up to 15 in our benchmark). For each query, we generate ground truth feasibility by running Gurobi on the query with its objective function removed. This will allow Gurobi to terminate as soon as it finds a feasible solution. We restrict the

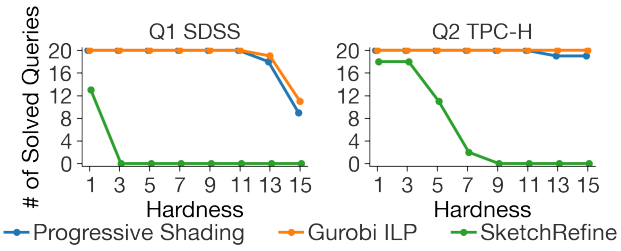

Figure 9: False infeasibility as hardness increases.

relation size to one million since this is the maximum size that Gurobi can solve within the time limit. Furthermore, for each dataset and hardness level, we randomly sample 20 sub-relations of size one million representing 20 queries and compute the number of queries for which each of the methods can find a solution.

Figure 9 displays our results. For Q1 SDSS, SketchRefine solves 13 out of 20 queries solved by Gurobi at $\tilde{h} = 1$ and none at all for $\tilde{h} > 1$. For Q2 TPC-H, SketchRefine solves only half as much for the usual workload of $\tilde{h} \in \{1, 3, 5\}$ but then fails to solve most of the hard queries where $\tilde{h} > 5$. On the other hand, Progressive Shading can solve almost as many as Gurobi solves in both queries. Results for the other queries we examined are similar; see [22, Appendix F.1] for details.

5 RELATED WORK

In-database optimization. Recent research aims to integrate complex analytics capabilities into DBMSs. SolveDB [33] provides extensible infrastructure for integrating a variety of black-box optimization solvers into a DBMS, whereas Progressive Shading focuses on ILP solvers and "opens up the black box" in order to scale to large problems. SolveDB offers built-in problem partitioning which is only applicable when there are sub-problems that can be solved independently, i.e., constraints that only exist within each sub-problem. However, such a partitioning strategy is ineffective for solving package queries because most of the tuples can connect via a single constraint and thus cannot be partitioned further. DLV provides a simple solution to the specific needs of package queries by partitioning a very large relation into similar tuples.

Resource allocation problems. Partitioned Optimization Problems (POP) [25] is a recent technique to solve large-scale "granular" resource allocation problems that can be often formulated as ILPs whose structures are different from ILPs formulated by package queries, i.e., the number of constraints in POP can be as large as the number of variables [21]. POP achieves high scalability by randomly splitting the problem into sub-problems and aggregating the resulting sub-allocations into a global allocation—an approach similar to SketchRefine [5]. Thus, POP still suffers from the same disadvantages as SketchRefine when we increase the scale because the number of sub-problems is up to 32 in POP. Moreover, the partitioning in POP is online while Progressive Shading is a large-scale package query solver that runs on an offline partition produced by DLV.

Semantic window queries. Semantic windows [16] are related to packages. A semantic window refers to a subset of a grid-partitioned space that is contiguous and has certain global properties. For example, astronomers may divide the night sky into a grid and search for areas where the overall brightness exceeds a particular threshold. Semantic windows can be expressed by package queries with a global condition to ensure that all cells in the package are contiguous. Searchlight [17], a recent method for answering semantic window queries, uses in-memory synopses to quickly estimate aggregate values of contiguous regions. This approach is analogous to our hierarchical partitioning strategy using DLV where a relation in layer l aggregates tuples from a relation in layer $l - 1$. However, Searchlight enumerates *all* of its feasible solutions and retains the best one—a very expensive computation—whereas Progressive Shading efficiently finds potentially optimal solutions via LP.

Neural Diving. Neural Diving [24] is a machine learning-based approach to solving ILPs that trains a deep neural network to produce multiple partial assignments of variables in the input ILP, with the remaining unassigned variables defining smaller sub-ILPs that can be solved using a black-box ILP solver. The neural network is trained on all available feasible assignments to give a higher probability to the ones that have better objective values instead of only the optimal ones, which can be expensive to collect. The authors of [24] evaluate the method on diverse datasets containing large-scale MIPs from real-world applications such as Google Production Planning, Electric Grid Optimization [19], and so on.

Unlike Dual Reducer, Neural Diving does not prune variables using an auxiliary LP but instead uses a pre-trained neural network. This approach requires expensive training over a large dataset of similar problem instances in order to learn effective heuristics. Moreover, solving package queries beyond millions is currently out of reach for Neural Diving since it requires the neural network—whose size scales with the number of variables and constraints—to fit in memory.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we expand our ability significantly beyond prior art [5] to solve challenging package queries over very large relations. Our novel Progressive Shading strategy uses a hierarchy of relations created via a sequence of partitionings, smartly *augments* the size of ILP subproblems to avoid false infeasibility, and provides a novel mechanism to parallelize and reduce solving time.

In future work, we plan to investigate combining Neural Diving and Progressive Shading to potentially solve a wide range of ILP problems (not just package queries) in arbitrarily large relations. Although Neural Diving is not currently a feasible approach, running large-scale neural networks inside a DBMS will eventually become efficient, e.g., by integrating tensor technology into DBMS [6, 13]. Then—because it is straightforward to generate different package queries with various hardnesses and query structures—a potential approach for solving package queries with high hardness would train Neural Diving using the feasible solutions generated from Dual Reducer.

ACKNOWLEDGMENTS

This work was supported by the ASPIRE Award for Research Excellence (AARE-2020) grant AARE20-307 and NYUAD CITIES, funded by Tamkeen under the Research Institute Award CG001, and by the National Science Foundation under grants 1943971 and 2211918.

REFERENCES

[1] Abdurro'uf and et al. 2021. The Seventeenth Data Release of the Sloan Digital Sky Surveys: Complete Release of MaNGA, MaStar and APOGEE-2 Data. arXiv:2112.02026. https://doi.org/10.3847/1538-4365/ac4414

[2] V. A. Alegana, P. M. Atkinson, C. Pezzulo, A. Sorichetta, D. Weiss, T. Bird, E. Erbach-Schoenberg, and A. J. Tatem. 2015. Fine resolution mapping of population age-structures for health and development applications. *Journal of The Royal Society Interface* 12, 105 (April 2015), 20150073. https://doi.org/10.1098/rsif.2015.0073

[3] Timo Berthold. 2009. *RENS-relaxation enforced neighborhood search.* Technical Report. Zuse Institute Berlin (ZIB).

[4] Robert Bixby and Alexander Martin. 2000. Parallelizing the Dual Simplex Method. *INFORMS Journal on Computing* 12 (02 2000), 45–56. https://doi.org/10.1287/ijoc.12.1.45.11902

[5] Matteo Brucato, Azza Abouzied, and Alexandra Meliou. 2018. Package queries: efficient and scalable computation of high-order constraints. *The VLDB Journal* 27, 5 (01 Oct 2018), 693–718. https://doi.org/10.1007/s00778-017-0483-4

[6] Wei Cui, Qianxi Zhang, Spyros Blanas, Jesús Camacho-Rodríguez, Brandon Haynes, Yinan Li, Ravi Ramamurthy, Peng Cheng, Rathijit Sen, and Matteo Interlandi. 2023. Query Processing on Gaming Consoles. In *Proceedings of the 19th International Workshop on Data Management on New Hardware* (Seattle, WA, USA) *(DaMoN '23)*. Association for Computing Machinery, New York, NY, USA, 86–88. https://doi.org/10.1145/3592980.3595313

[7] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.

[8] Raphael Finkel and Jon Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4 (03 1974), 1–9. https://doi.org/10.1007/BF00288933

[9] Matteo Fischetti and Andrea Lodi. 2011. Heuristics in Mixed Integer Programming. https://doi.org/10.1002/9780470400531.eorms0376

[10] Gurobi Optimization, LLC. 2022. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[11] Boukthir Haddar, Mahdi Khemakhem, Saïd Hanafi, and Christophe Wilbaut. 2015. A hybrid heuristic for the 0–1 Knapsack Sharing Problem. *Expert Systems with Applications* 42, 10 (June 2015), 4653–4666. https://doi.org/10.1016/j.eswa.2015.01.049

[12] J. A. Hartigan and M. A. Wong. 1979. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979), 100–108. http://www.jstor.org/stable/2346830

[13] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2811–2825. https://doi.org/10.14778/3551793.3551833

[14] Frederick S. Hillier. 1967. *Introduction to Operations Research.* San Francisco, Holden-Day.

[15] Q. Huangfu and J. A. J. Hall. 2018. Parallelizing the dual revised simplex method. *Mathematical Programming Computation* 10, 1 (01 Mar 2018), 119–142. https://doi.org/10.1007/s12532-017-0130-5

[16] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. 2014. Interactive Data Exploration Using Semantic Windows. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 505–516. https://doi.org/10.1145/2588555.2593666

[17] Alexander Kalinin, Ugur Cetintemel, and Stan Zdonik. 2015. Searchlight: Enabling Integrated Search and Exploration over Large Multidimensional Data. *Proc. VLDB Endow.* 8, 10 (jun 2015), 1094–1105. https://doi.org/10.14778/2794367.2794378

[18] Leonard Kaufman and Peter J. Rousseeuw. 1990. *Finding Groups in Data: An Introduction to Cluster Analysis.* John Wiley.

[19] Bernard Knueven, James Ostrowski, and Jean-Paul Watson. 2020. On Mixed-Integer Programming Formulations for the Unit Commitment Problem. *INFORMS Journal on Computing* (June 2020). https://doi.org/10.1287/ijoc.2019.0944

[20] C. C. N. Kuhn, G. Calbert, I. Garanovich, and T. Weir. 2023. Integer linear programming supporting portfolio design. arXiv:2303.14364 [math.OC]

[21] Xiaoqian Li and Kwan L. Yeung. 2019. Traffic Engineering in Segment Routing using MILP. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. IEEE. https://doi.org/10.1109/icc.2019.8762075

[22] Anh Mai, Matteo Brucato, Azza Abouzied, Peter J. Haas, and Alexandra Meliou. 2023. Scaling Package Queries to a Billion Tuples via Hierarchical Partitioning and Customized Optimization. https://github.com/alm818/PackageQuery. arXiv:2307.02860 [cs.DB] https://doi.org/10.48550/arXiv.2307.02860

[23] Rajesh Matai, Surya Singh, and Murari Lal Mittal. 2010. Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches. In *Traveling Salesman Problem*, Donald Davendra (Ed.). IntechOpen, Rijeka, Chapter 1. https://doi.org/10.5772/12909

[24] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, Ravichandra Addanki, Tharindi Hapuarachchi, Thomas Keck, James Keeling, Pushmeet Kohli, Ira Ktena, Yujia Li, Oriol Vinyals, and Yori Zwols. 2020. Solving Mixed Integer Programs Using Neural Networks. https://doi.org/10.48550/ARXIV.2012.13349

[25] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, Akshay Agrawal, Srikanth Kandula, Stephen P. Boyd, and Matei Zaharia. 2021. Solving Large-Scale Granular Resource Allocation Problems Efficiently with POP. *CoRR* abs/2110.11927 (2021). arXiv:2110.11927 https://arxiv.org/abs/2110.11927

[26] J.C. Nash. 2000. The (Dantzig) simplex method for linear programming. *Computing in Science & Engineering* 2, 1 (2000), 29–31. https://doi.org/10.1109/5992.814654

[27] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. 2018. Vesper: Measuring Time-to-Interactivity for Web Pages. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 217–231. https://www.usenix.org/conference/nsdi18/presentation/netravali-vesper

[28] Michael J. Panik. 1996. *The Dual Simplex, Primal-Dual, and Complementary Pivot Methods.* Springer US, Boston, MA, 251–288. https://doi.org/10.1007/978-1-4613-3434-7_10

[29] Maurice Roux. 2015. A comparative study of divisive hierarchical clustering algorithms. *CoRR* abs/1506.08977 (2015). arXiv:1506.08977 http://arxiv.org/abs/1506.08977

[30] Georg Sander and Adrian Vasiliu. 2005. Visualization and ILOG CPLEX. In *Graph Drawing*, János Pach (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 510–511.

[31] TPCH [n.d.]. TPC-H Decision Support Benchmark. https://www.tpc.org/tpch/.

[32] Vijay V. Vazirani. 2003. *Approximation Algorithms.* Springer Berlin Heidelberg, 108. https://doi.org/10.1007/978-3-662-04565-7

[33] Laurynas Šikšnys and Torben Bach Pedersen. 2016. SolveDB: Integrating Optimization Problem Solvers Into SQL Databases. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management* (Budapest, Hungary) *(SSDBM '16)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/2949689.2949693