# *Stochastic SketchRefine*: Scaling In-Database Decision-Making under Uncertainty to Millions of Tuples

Riddho R. Haque
University of Massachusetts Amherst
rhaque@cs.umass.edu

Anh L. Mai
NYU Abu Dhabi
anh.mai@nyu.edu

Matteo Brucato
Microsoft Research
mbrucato@microsoft.com

Azza Abouzied
NYU Abu Dhabi
azza@nyu.edu

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

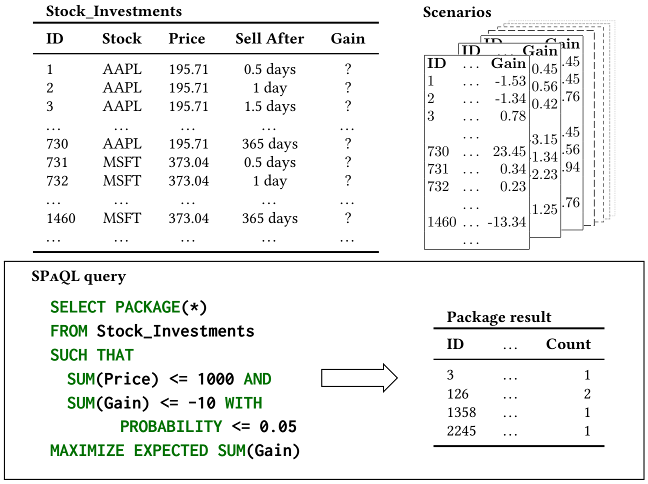
Alexandra Meliou
University of Massachusetts Amherst
ameli@cs.umass.edu

## ABSTRACT

Decision making under uncertainty often requires choosing *packages*, or bags of tuples, that collectively optimize expected outcomes while limiting risks. Processing *Stochastic Package Queries* (SPQs) involves solving very large optimization problems on uncertain data. Monte Carlo methods create numerous *scenarios*, or sample realizations of the stochastic attributes of *all* the tuples, and generate packages with optimal objective values across these scenarios. The number of scenarios needed for accurate approximation—and hence the size of the optimization problem when using prior methods—increases with variance in the data, and the search space of the optimization problem increases exponentially with the number of tuples in the relation. Existing solvers take hours to process SPQs on large relations containing stochastic attributes with high variance. Besides enriching the SPaQL language to capture a broader class of risk specifications, we make two fundamental contributions toward scalable SPQ processing. First, we propose *risk-constraint linearization* (RCL), which converts SPQs into Integer Linear Programs (ILPs) whose size is independent of the number of scenarios used. Solving these ILPs gives us feasible and near-optimal packages. Second, we propose Stochastic SketchRefine, a divide and conquer framework that breaks down a large stochastic optimization problem into subproblems involving smaller subsets of tuples. Our experiments show that, together, RCL and Stochastic SketchRefine produce high-quality packages in orders of magnitude lower runtime than the state of the art.

**PVLDB Reference Format:**
Riddho R. Haque, Anh L. Mai, Matteo Brucato, Azza Abouzied, Peter J. Haas, Alexandra Meliou. *Stochastic SketchRefine*: Scaling In-Database Decision-Making under Uncertainty to Millions of Tuples. PVLDB, 18(9): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at github.com/RiddhoHaque/stochastic-sketchrefine.



**Stock_Investments**

| ID | Stock | Price | Sell After | Gain |
|---|---|---|---|---|
| 1 | AAPL | 195.71 | 0.5 days | ? |
| 2 | AAPL | 195.71 | 1 day | ? |
| 3 | AAPL | 195.71 | 1.5 days | ? |
| … | … | … | … | … |
| 730 | AAPL | 195.71 | 365 days | ? |
| 731 | MSFT | 373.04 | 0.5 days | ? |
| 732 | MSFT | 373.04 | 1 day | ? |
| … | … | … | … | … |
| 1460 | MSFT | 373.04 | 365 days | ? |
| … | … | … | … | … |

**Scenarios**

| ID | … | Gain |
|---|---|---|
| 1 | … | -1.53 |
| 2 | … | -1.34 |
| 3 | … | 0.78 |
| | … | |
| 730 | … | 23.45 |
| 731 | … | 0.34 |
| 732 | … | 0.23 |
| | … | |
| 1460 | … | -13.34 |
| | … | |

**SPaQL query**

```
SELECT PACKAGE(*)
FROM Stock_Investments
SUCH THAT
  SUM(Price) <= 1000 AND
  SUM(Gain) <= -10 WITH
       PROBABILITY <= 0.05
MAXIMIZE EXPECTED SUM(Gain)
```

**Package result**

| ID | … | Count |
|---|---|---|
| 3 | … | 1 |
| 126 | … | 2 |
| 1358 | … | 1 |
| 2245 | … | 1 |

**Figure 1: The gain in the Stock_Investments table is an uncertain attribute, evaluated via simulated stochastic processes. The scenarios represent different simulations (possible worlds). The example SPaQL query contains a value-at-risk (VaR) constraint, specifying that the probability of total loss (negative gain) exceeding $10 is at most 5%.**

## 1 INTRODUCTION

Many decision-making problems involve risk-constrained optimization over uncertain data [2]. Consider a stock portfolio optimization problem (Figure 1), where stock prices at future dates are uncertain and are simulated as stochastic processes (e.g., Geometric Brownian Motion [33]). We choose which stocks to buy, how many shares of each, and when to sell them, with constraints on budget and risk of loss. Such a problem can be expressed as a package query using SPaQL—Stochastic Package Query Language—allowing for the benefits of in-database decision-making [6]. The example query in Figure 1 requests a portfolio that costs less than $1000 and for which the probability of losing more than $10 is at most 5%—the latter constraint is called a *Value-at-Risk* (VaR) constraint. The package result is a bag of tuples, i.e., a portfolio of stocks and a selling schedule, that satisfies the constraints while maximizing the expected gain.

To solve the stochastic optimization problem specified by a SPaQL query, we approximate it by a deterministic problem via a Monte Carlo technique called Sample Average Approximation (SAA) [27]. SAA creates numerous scenarios ("possible worlds"), each comprising a sample realization for every stochastic attribute

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:XX.XX/XXX.XX

of every tuple in the relation, e.g., a realized gain for every Stock–Sell_After pair as in Figure 1. Such scenarios can be created using the Variable Generation (VG) functions of Monte Carlo databases like MCDB [26]. A VG function is a user-defined function that takes a table of input parameters and generates a table—a scenario—of sample values drawn from a corresponding probability distribution. Because of their flexibility, VG functions can capture complex correlations across the sampled tuples. In Figure 1, a VG function can use simulation models trained on historical stock prices to simulate discretized trajectories of future prices of every stock, and create scenarios of potential gains.

The scenarios are used to construct an SAA approximation to the original SPQ, where expectations are replaced by averages over scenarios and probabilities are replaced by empirical probabilities. E.g., the SAA approximation to the example query in Figure 1 maximizes the average profit over all the scenarios, with at most 5% of the scenarios having losses over $10.

Stochastic Package Queries (SPQs) are hard to scale on large datasets. They represent constrained optimization problems that grow along both: (i) *the tuple dimension* and (ii) *the scenario dimension.* More tuples directly correspond to more decision variables in an SPQ, causing the search space of the optimization problem—and correspondingly the solver runtime—to grow exponentially. On the other hand, increasing the number of scenarios is necessary for SAA accuracy, especially when stochastic attributes have high variance. However, this also increases the runtime for creating these scenarios, formulating and solving SAA optimization problems (whose size grows with the number of scenarios), and assessing the empirical risks of packages over the scenario set. Even worse, having more tuples also necessitates having more scenarios due to the curse of dimensionality—more samples are needed to make the SAA sufficiently accurate in high-dimensional decision spaces [11].

**Prior work** tackled scaling along either the tuple or the scenario dimension, but not both. For all methods (including ours), we assume that the size of the optimal package is small, which is often the case for real-world problems. SketchRefine [4] efficiently processes deterministic package queries (no scenarios involved) on large relations by partitioning the relations into groups of similar tuples and constructing representatives of each group offline. During query execution, the *sketch* phase solves the problem over only the representatives; the *refine* phase then iteratively replaces representatives in the sketch package with tuples from their partitions. Both sketch and refine queries are small enough ILPs that can be directly solved by off-the-shelf optimizers such as Gurobi.

SummarySearch [6] helps scale SPQ processing along the scenario dimension. It exploits the fact that SPQs can be approximated as ILPs using SAA. It creates packages using a set of *optimization scenarios* and validates their feasibility over a much larger set of *validation scenarios.* The key challenge is that large numbers of optimization scenarios are typically needed to mitigate the "optimizer's curse", which occurs when packages created from the optimization scenarios violate risk constraints among the validation scenarios; however, the size—and hence processing time—of the ILPs grows with the number of optimization scenarios. To handle this scaling issue, it replaces the optimization scenarios by a small set of conservative scenarios called *summaries.* Constraints based on these summaries are harder to satisfy than constraints based on a random set of scenarios as in SAA, so packages created from them are more risk-averse and likely to be validation-feasible. E.g., the summary of a set of scenarios in our investment example might comprise the scenario-wise minimum gain for each tuple. By varying the number of summaries, conservativeness can be carefully controlled to ensure near-optimal and feasible solutions.

These methods have serious limitations. SketchRefine does not work on stochastic data, and SummarySearch does not scale well with more tuples because the formulated ILP has one decision variable per tuple in the relation. Moreover, SummarySearch does not work well with high-variance stochastic attributes: As variance increases, more optimization scenarios are needed to accurately reflect the properties of the validation scenarios. With too few optimization scenarios, an intermediate package is likely to be validation-feasible but sub-optimal, so SummarySearch will spend a lot of time increasing the number of summaries to relax the problem, only to discover that more scenarios are needed.

**Our work** both enriches the expressiveness of the SPaQL language and is the first to scale SPQ processing along both the tuple and the scenario dimensions. With respect to expressiveness, we extend SPaQL to allow specification of *Conditional Value-at-Risk* (CVaR) constraints, also called "expected shortfall" constraints. The CVaR risk measure is widely used in finance, insurance, and other risk-management domains [3, 29, 34]. The VaR constraint in our portfolio example is of the form:

```
SUM(Gain)<= -10 WITH PROBABILITY <= 0.05
```

whereas a CVaR constraint might be of the form:

```
EXPECTED SUM(Gain)>= -10 IN LOWER 0.05 TAIL
```

Roughly speaking, the former constraint requires that the "bad event" where the loss exceeds $10 occur with probability of at most 0.05, whereas the latter requires that, given the occurrence of a high-loss event of the form "loss exceeds $$x$" having 0.05 probability, the expected value of the loss does not exceed $10. (Section 2 gives precise definitions.) Though VaR is a popular risk measure, risk analysts often prefer CVaR over VaR because it is a *coherent* risk measure with nice mathematical properties [34]. E.g., portfolio optimization problems with CVaR constraints promote diversification of stocks, and CVaR can avoid extreme losses more effectively than VaR. For example, given a portfolio that satisfies the VaR constraint, if an investor loses at least $10 (which happens with at most 5% probability), they can actually lose much more than $10 without any way to control the expected loss. Having an additional CVaR constraint caps such expected losses when the bad event occurs.

We achieve scaling through two novel mechanisms: *risk-constraint linearization* (RCL) and Stochastic SketchRefine.

RCL handles scenario-scaling issues by replacing each VaR and CVaR constraint by a *linearized CVaR* (L-CVaR) constraint (Section 2). Crucially, an SPQ with only L-CVaR constraints leads to a smaller SAA approximation whose size is independent of the number of scenarios and can be solved efficiently. Like CVaR, an L-CVaR constraint is parameterized by two values: a tail specifier $\alpha$ and a value $V$, e.g., $\alpha = 0.05$ and $V = -10$ in the foregoing CVaR example. Our new RCL procedure efficiently finds values of $\alpha$ and $V$ for each constraint such that the resulting feasible region for the SAA problem contains a solution that is both feasible and near optimal with respect to original SPQ.

Stochastic SketchRefine handles tuple-scaling issues by using a new data partitioning and evaluation mechanism. It retains the divide-and-conquer structure of SketchRefine—similarly leveraging off-the-shelf solvers and using sets of optimization and validation scenarios—but incorporates non-trivial extensions to overcome challenges raised by stochasticity. Stochastic SketchRefine uses our novel, trivially-parallelizable partitioning method, DistPartition. Unlike existing hierarchical stochastic data clustering approaches [21, 25], DistPartition's time complexity is sub-quadratic with respect to the number of tuples in the relation. This makes it suitable for partitioning million-tuple relations.

**Contributions and outline.** RCL and Stochastic SketchRefine synergistically reduce overall latencies, ensuring superior scalability over both tuples and scenarios. Stochastic SketchRefine keeps high-variance tuples in separate partitions due to their dissimilarity with low-variance tuples. While refining partitions with high-variance tuples, use of RCL reins in the SPQ-processing latencies. In more detail, we organize our contributions as follows.

- We extend SPaQL to allow for the expression of CVaR constraints, which allow for better risk control in stochastic optimization problems. We further discuss necessary background and provide an overview of our approach and key insights. [Section 2]
- We detail how RCL replaces VaR and CVaR constraints in an SPQ by L-CVaR constraints, while ensuring the resulting packages are feasible and near-optimal for the original SPQ. [Section 3]
- We present Stochastic SketchRefine, a two-phase divide-and-conquer approach that splits SPQs on large relations into smaller problems that can be solved quickly. Similar to SketchRefine, it first solves a *sketch* problem using representatives over data partitions, and then *refines* the initial solution with data from each partition. In contrast with SketchRefine, our method employs stochastically-identical *duplicates* of representatives to handle the challenges of stochasticity. [Section 4]
- Stochastic SketchRefine relies on appropriately-partitioned data to generate the sketch and refine problems. We propose a novel offline partitioning method for stochastic relations, DistPartition, which can effectively partition stochastic relations containing millions of tuples in minutes. [Section 5]
- We show that Stochastic SketchRefine achieves a $(1-\epsilon)^2$-optimal solution w.r.t a user-defined approximation error bound $\epsilon$. [23, Appendix E]
- We show via experiments that Stochastic SketchRefine generates high quality packages in an order of magnitude lower runtime than SummarySearch. Furthermore, Stochastic SketchRefine can execute package queries over millions of tuples within minutes, whereas SummarySearch fails to produce any package at such scales even after hours of execution. [Section 6]

## 2 CVAR AND RELATED CONSTRAINTS

In this section, we discuss the extension of SPaQL via the addition of CVaR constraints. We first briefly review the types of constraints previously supported by SPaQL and then describe the syntax and semantics of CVaR and related constraints in detail.

**SPaQL constraint modeling.** We assume a relation with $n$ tuples. In SPQ evaluation, each tuple $t_i$ is associated with an integer variable $x_i$ that represents the tuple's multiplicity in the package result. A *feasible package* is an assignment of the variables in $x = (x_1, \dots, x_n)$, that satisfies all query constraints; as in the introductory example, we typically want to find a feasible package that maximizes or minimizes a given linear objective function; see [7, Appendix A] for a complete SPaQL language description. We denote by $t_i.A$ the value of the attribute $A$ in tuple $t_i$; if $A$ is stochastic, then $t_i.A$ is a random variable. The SPaQL constraint types are as follows.

- `REPEAT R` : *Repeat constraints* cap the multiplicities of every tuple in the package, i.e., $x_i \le (1+R), \forall 1 \le i \le n$.
- `COUNT(Package.*)<= S` : *Package-size constraints* bound the total number of tuples in the package, i.e., $\sum_{i=1}^{n} x_i \le S$.
- `SUM(A)<= V` : *Deterministic sum constraints* bound the sum of the values of a deterministic attribute $A$ in the package, i.e., $\sum_{i=1}^{n} t_i.A * x_i \le V$.
- `EXPECTED SUM(A)<= V` : *Expected sum constraints* bound the expected value of the sum of the values of a stochastic attribute $A$ in the package, i.e., $\mathbb{E}[\sum_{i=1}^{n} t_i.A * x_i] \le V$.
- `SUM(A)<= V WITH PROBABILITY <= α` : *VaR constraints* bound the probability that the sum of stochastic attribute $A$ is below or above a value, i.e., $\mathbb{P}(\sum_{i=1}^{n} t_i.A * x_i \le V) \le \alpha$.

**Value-at-Risk.** We first define the notions of quantile and VaR. Consider a stochastic attribute $A$ with cumulative distribution function $F_A$. For $\alpha \in [0,1]$, we define the *$\alpha$-quantile* of $A$—or, equivalently, of $F_A$—as $q_\alpha(A) = \inf\{y : F_A(y) \ge \alpha\}$.We can then define the *$\alpha$-confidence Value at Risk* of $A$ as $\mathrm{VaR}_\alpha(A) = q_\alpha(A)$,i.e., the VaR is simply a quantile. Thus a VaR constraint as above can be interpreted as a constraint of the form $\mathrm{VaR}_\alpha(\sum_{i=1}^{n} t_i.A * x_i) \ge V$.

**Conditional Value-at-Risk.** As discussed in the introduction, CVaR constraints help control extreme risks beyond what VaR constraints can provide. Following [29, 34], we define the *lower-tail $\alpha$-confidence Conditional Value-at-Risk* of $A$ as

$$\mathrm{CVaR}_\alpha^\perp(A) = \frac{1}{\alpha}\int_0^\alpha q_u(A)\,du = \frac{1}{\alpha}\int_0^\alpha \mathrm{VaR}_u(A)\,du. \qquad (1)$$

and the *upper-tail $\alpha$-confidence Conditional Value-at-Risk* of $A$ as

$$\mathrm{CVaR}_\alpha^\top(A) = \frac{1}{1-\alpha}\int_\alpha^1 q_u(A)\,du = \frac{1}{1-\alpha}\int_\alpha^1 \mathrm{VaR}_u(A)\,du.$$

From [29, Lemma 2.16], we have that if $F_A$ is continuous, then

$$\mathrm{CVaR}_\alpha^\perp(A) = \mathbb{E}[A \mid A \le q_\alpha(A)] \qquad (2)$$

$$\text{and} \qquad \mathrm{CVaR}_\alpha^\top(A) = \mathbb{E}[A \mid A \ge q_\alpha(A)], \qquad (3)$$

which motivates our CVaR-constraint syntax. Specifically, for a value such as $\alpha = 0.05$, the following SPaQL CVaR constraints

```
EXPECTED SUM(A) >= V IN LOWER α TAIL
EXPECTED SUM(A) <= V IN UPPER α TAIL
```

correspond to the constraints $\mathrm{CVaR}_\alpha^\perp(\sum_{i=1}^{n} t_i.A * x_i) \ge V$ and $\mathrm{CVaR}_{1-\alpha}^\top(\sum_{i=1}^{n} t_i.A * x_i) \le V$, respectively. In a portfolio setting, constraints using $\mathrm{CVaR}_\alpha^\perp(A)$ are useful when $A$ represents a gain as in our portfolio example, and constraints using $\mathrm{CVaR}_{1-\alpha}^\top(A)$ are useful when $A$ represents a loss. Since $\mathrm{CVaR}_\alpha^\perp(A) = \mathrm{CVaR}_{1-\alpha}^\top(-A)$, without loss of generality, we will focus on $\mathrm{CVaR}_\alpha^{\perp}$ and simply denote it by $\mathrm{CVaR}_\alpha$; we will also restrict attention to maximization problems where the use of $\mathrm{CVaR}_\alpha^{\perp}$ makes sense. Also, for simplicity, we assume henceforth that all random attributes have continuous distributions so that the representations in (2) and (3) are valid. Then, given a set $S$ of i.i.d. samples from the distribution $F_Z$ of a random variable $Z$, we can estimate $\mathrm{CVaR}_\alpha(Z)$ simply as $\widehat{\mathrm{CVaR}}_\alpha(Z)$, the average over the lowest $\alpha$-fraction of values in $S$.[1]

**Scalably solving SPQs with linearized CVaR.** In contrast with VaR constraints, CVaR constraints are convex, naturally reducing the complexity of SPQs. However, replacing each VaR constraint with CVaR—so the query contains only CVaR constraints—does not suffice to solve SPQs efficiently: The convex optimization problem resulting from the modified SPQ is not amenable to SAA approximation and is usually very expensive to solve. To scalably solve SPQs, we introduce the notion of *linearized CVaR constraints.* For a package represented by integer variables[2] $x = (x_1, \ldots, x_n)$—i.e., the multiplicities of tuples in the package—and a stochastic attribute $A$, define

$$\text{L-CVaR}_\alpha(x, A) = \sum_{i=1}^{n} \mathrm{CVaR}_\alpha(t_i.A) * x_i.$$

An L-CVaR constraint has the form $\text{L-CVaR}_\alpha(x, A) \geq V$ (in the lower $\alpha$-tail). When identically parameterized, an L-CVaR constraint is more restrictive than a CVaR constraint, which in turn is more restrictive than a VaR constraint. Formally, letting $x \cdot A$ denote $\sum_{i=1}^{n} t_i.A * x_i$ and $\mathbb{Z}_0$ denote the set of nonnegative integers, we have the following result, which holds even if $F_A$ is not continuous. See [23, Appendix B] for all proofs.

Theorem 2.1. *For any $x \in \mathbb{Z}_0^n$, $\alpha \in [0, 1]$, $V \in \mathbb{R}$, and stochastic attribute $A$:*

$$\begin{aligned} \textit{L-CVaR}_\alpha(x, A) \geq V &\implies \textit{CVaR}_\alpha(x \cdot A) \geq V \\ &\implies \textit{VaR}_\alpha(x \cdot A) \geq V \end{aligned}$$

As a simple example, consider a relation with two tuples $t_1$ and $t_2$ and a stochastic attribute $A$ such that $\mathbb{P}(t_1.A = -1) = \mathbb{P}(t_1.A = 1) = 0.5$, and $t_2.A$ and $t_1.A$ are i.i.d. For $x = (1, 1)$ and $\alpha = 0.5$, we can verify that $\text{L-CVaR}_{0.5}(x, A) \leq \mathrm{CVaR}_{0.5}(x \cdot A) \leq \mathrm{VaR}_{0.5}(x \cdot A)$, which implies the assertion of the theorem for this example. Observe that $\mathrm{CVaR}_{0.5}(t_1.A) = \mathrm{CVaR}_{0.5}(t_2.A) = -1$, so that $\text{L-CVaR}_{0.5}(x, A) = 1 \cdot \mathrm{CVaR}_{0.5}(t_1.A) + 1 \cdot \mathrm{CVaR}_{0.5}(t_1.A) = -2$. Also, the random variable $Z = x \cdot A = 1 \cdot t_1.A + 1 \cdot t_2.A$ satisfies $\mathbb{P}(Z = -2) = \mathbb{P}(Z = 2) = 0.25$ and $\mathbb{P}(Z = 0) = 0.5$. Thus $\mathrm{CVaR}_{0.5}(x \cdot A) = \mathbb{E}[Z \mid Z \leq 0] = -2/3$ and $\mathrm{VaR}_{0.5}(x \cdot A) = \mathrm{median}(Z) = 0$, verifying the inequalities. Intuitively, the first inequality holds—strictly, in this example—because L-CVaR involves separate averages over the lower tails of $t_1.A$ and $t_2.A$, each of which have half their probability mass strictly below the median at 0. In contrast, the random variable $Z$ has significant mass at the median value of 0, because positive values of $t_1.A$ can compensate for negative values of $t_2.A$ and vice versa, so the CVaR lower-tail average is higher. Moreover, VaR is simply the median of $Z$ whereas CVaR is an average of values at or below the median, which explains the second (strict) inequality.

**Algorithm 1** RCL-Solve

```
Input: $Q$ := A Stochastic Package Query
       $S$ := Set of stochastic tuples in the SPQ
       $m$ := Initial number of optimization scenarios
       $\mathcal{V}$ := Set of validation scenarios
       $\delta$ := Bisection termination distance
       $\epsilon$ := Error bound
Output: $x$ := A validation-feasible and $\epsilon$-optimal package for $Q(S)$
            (or NULL if unsolvable)
$\mathcal{O} \leftarrow$ GenerateScenarios$(m)$                      ▹ Initial optimization scenarios
qsSuccess, $x^{\mathrm{D}} \leftarrow$QuickSolve(Q, S)                  ▹ Check if solution is "easy"
if qsSuccess = True then
    return $x^{\mathrm{D}}$                                    ▹ Either solution to $Q(S)$ or NULL
$\alpha, V \leftarrow$ GetParams$(Q)$                           ▹ Extract $(\alpha_r, V_r)$ for each $r \in Q$
$\omega_0 \leftarrow$ OmegaUpperBound$(Q, S, x^{\mathrm{D}}, \mathcal{V})$
while True do
    $V' \leftarrow V, \alpha' \leftarrow 1$                         ▹ Initial L-CVaR parameter values
    Compute $\hat{V}_0$ as in (4) and set $V_L \leftarrow \hat{V}_0, V_U \leftarrow V, \alpha_L \leftarrow \alpha, \alpha_U \leftarrow 1$
    while True do
        ▹ Search for $\alpha$
        $\alpha'_{\mathrm{old}} \leftarrow \alpha'$
        status, $\alpha', x \leftarrow \alpha$-Search$(V', \alpha_L, \alpha_U, Q, S, \epsilon, \delta, \mathcal{O}, \mathcal{V}, \omega_0)$
        if status.NeedScenarios = True then break
        if status.Done = True then return $x$
        ▹ Search for $V$
        $V'_{\mathrm{old}} \leftarrow V'$
        status, $V', x \leftarrow$ V-Search$(\alpha', V_L, V_U, Q, S, \epsilon, \delta, \mathcal{O}, \mathcal{V}, \omega_0)$
        if status.NeedScenarios = True then break
        if status.Done = True then return $x$
        $V_U \leftarrow \max(V' - \delta, V_L)$
        if $\max(V'_{\mathrm{old}} - V', \alpha'_{\mathrm{old}} - \alpha') < \delta$ then break
    if $2m > |\mathcal{V}|$ then return $x$                      ▹ Best solution found so far
    $\mathcal{O} \leftarrow \mathcal{O} \cup$ GenerateScenarios$(m)$                 ▹ Double # of scenarios
    $m \leftarrow 2m$
```

## 3 RCL-SOLVE: SOLVING SMALL SPQS

In this section, we describe RCL-Solve (Algorithm 1), a standalone scenario-scalable algorithm for solving SPQs over a relatively small set of tuples. By completely eliminating the need for scenario summaries, RCL-Solve outperforms SummarySearch. Moreover, our new Stochastic SketchRefine algorithm (Section 4) solves a large-scale SPQ with many tuples by solving a sequence $Q(S_0), Q(S_1), \ldots$ of relatively small-scale SPQs, and slight variants of Algorithm 1 (discussed in Section 4) can be used to solve each SPQ encountered during the sketch-and-refine process.

**Overview.** The basic idea is to replace each *risk constraint* (VaR or CVaR constraint) in an SPQ with a corresponding L-CVaR constraint of the form $\text{L-CVaR}_\alpha(x, A) \geq V$. Such replacement ensures that the resulting SAA problem size is independent of the number of scenarios. Moreover, the feasible region (in the space of possible packages), which was formerly non-linear and non-convex, is transformed to a convex polytope, so that the modified SAA problem can be efficiently solved using an ILP solver. We carefully choose the L-CVaR constraints so that (1) the package solution for the reshaped feasible region is *validation-feasible*, i.e., satisfies the *original* set of risk constraints with respect to the validation scenarios, and (2) the package solution is *$\epsilon$–optimal*, i.e., its objective value is $\geq (1-\epsilon)\omega$, where $\omega$ is the objective value for the true optimal solution to the original SPQ and $\epsilon$ is an application-specific error tolerance. Since the value of $\omega$ is unknown, we use an upper bound $\overline{\omega} \geq \omega$, so that any package with objective value above $(1-\epsilon)\overline{\omega}$ will have an objective value above $(1-\epsilon)\omega$ (see below). One simple choice for $\overline{\omega}$ that works well in practice is $\omega_0$—the objective value of the package solution $x^{\mathrm{D}}$ to query $Q^{\mathrm{D}}(S)$, where $Q^{\mathrm{D}}(S)$ is the deterministic package

[1] In general, $\mathrm{CVaR}_\alpha^{\perp}(Z) = \mathbb{E}[Z \mid Z \leq q_\alpha(Z)] + q_\alpha(Z)\big[\alpha - \mathbb{P}\big(Z \leq q_\alpha(Z)\big)\big]$, so the only modification to the methods in this paper is that estimation of CVaR from optimization scenarios becomes slightly more complex.

[2] In a slight abuse of terminology, we use the term "package" to refer interchangeably to either the integer vector $x$ or the bag of tuples specified by $x$.

query obtained by removing all probabilistic constraints from $Q(S)$, i.e., removing all VaR, CVaR and expected sum constraints (line 6).

*Risk-constraint linearization* (RCL) is the process of replacing all risk constraints in a query $Q(S)$ by L-CVaR constraints to achieve objectives (1) and (2) above. RCL is accomplished via a search over potential $(\alpha, V)$ values for every L-CVaR constraint. For each choice of values, we solve the resulting SPQ via SAA approximation using the optimization scenarios and then check whether the returned package is validation-feasible and $\epsilon$−optimal (as conservatively estimated above).

To design an effective search strategy, we first observe that, by Theorem 2.1, an L-CVaR constraint with parameters $\alpha$ and $V$ is more restrictive than a risk constraint with the same parameters. This strict L-CVaR parameterization, when applied to all constraints, results in a feasible-region polytope that is likely too small and may not contain the true optimal solution. Thus, the resulting package solution, while likely validation-feasible, will also likely be far from $\epsilon$−optimal. By varying the $\alpha$ and $V$ parameters, we can systematically shift and rotate the L-CVaR constraint boundaries until the feasible region contains a solution that is validation-feasible and $\epsilon$-optimal. As discussed below, for each constraint it suffices to search for optimal parameters in a bounded set of the form $[\alpha, 1] \times [V_0, V]$, where $\alpha$ and $V$ are the parameters of the original VaR or CVaR constraint, which lets us use an efficient alternating-parameter search algorithm that navigates between solutions that are suboptimal and solutions that are validation-infeasible to find the desired package solution. The user will know whether the parameter search was successful or not. If successful, RCL-Solve will return a validation-feasible and $\epsilon$−optimal package; otherwise, it will return the best validation-feasible package encountered so far, but with no optimality guarantees. Although the latter behavior is theoretically possible, we did not encounter any failed parameter searches in our experiments.

**Quick solution in special cases.** Sometimes a query $Q(S)$ can be solved quickly, without going through the entire RCL process. If the ILP solver can find a package solution $x^{\mathrm{D}}$ to query $Q^{\mathrm{D}}(S)$ as above, and if this solution is validation-feasible for $Q(S)$, then $x^{\mathrm{D}}$ is the solution to $Q(S)$, since it satisfies all constraints, including the probabilistic ones, and the objective value is as high as possible since the probabilistic constraints have been completely relaxed, thereby maximizing the feasible region. On the other hand, if no feasible solution can be found for $Q^{\mathrm{D}}(S)$, then $Q(S)$ is also unsolvable, since $Q$ has all the constraints that $Q^{\mathrm{D}}$ has, and more. We denote by QuickSolve$(Q, S)$ the subroutine that detects these two possible outcomes (line 2). QuickSolve$(Q, S)$ returns a pair (qsSuccess, $x^{\mathrm{D}}$). If qsSuccess = True, then $x^{\mathrm{D}}$ either is the package solution to $Q(S)$ or $x^{\mathrm{D}}$ = NULL, which indicates that $Q(S)$ is unsolvable, and RCL-Solve terminates and returns one of these values (lines 3–4). If qsSuccess = False then $x^{\mathrm{D}}$ solves $Q^{\mathrm{D}}(S)$ but is validation-infeasible for $Q(S)$ and the full RCL process is needed.

**The case of a single risk constraint.** To understand how RCL works in the case that qsSuccess = False, first consider an SPQ $Q(S)$ having a single risk constraint—on a random attribute $A$ with parameters $(\alpha, V)$—that will be replaced by an L-CVaR constraint. If the L-CVaR constraint is too stringent, the package solution will be validation-feasible, but sub-optimal, whereas if the constraint is too relaxed, the package solution will be validation-infeasible. A package solution may also be validation-infeasible or sub-optimal if the number of optimization scenarios is too small so that the SAA approximation is not accurate. We will discuss how RCL deals with this latter issue shortly, but suppose for now that the initial number $m$ of optimization scenarios is adequate.

Let $\alpha'$ and $V'$ denote the adjusted parameters of the L-CVaR constraint. Recall from Theorem 2.1 that, for any risk constraint with parameters $\alpha$ and $V$, setting $\alpha' = \alpha$ and $V' = V$ will cause the corresponding L-CVaR constraint to be overly restrictive. Also note that an L-CVaR constraint of the form $\text{L-CVaR}_{\alpha'}(x, A) \geq V'$ becomes less restrictive as either $V'$ decreases or $\alpha'$ increases. The parameter $\alpha'$ can be set as high as 1, in which case each coefficient $\hat{C}_i = \text{L-CVaR}_{\alpha'}(t_i.A)$ will be the average of the $t_i.A$ values across *all* the optimization scenarios. On the other hand, for a given value of $\alpha$, the parameter $V'$ can be set as low as $V_0$, where $V_0 = \sum_{i=1}^{n} \text{CVaR}_\alpha(t_i.A) * x_i^D$ and $x^D = (x_1^D, \ldots, x_n^D)$ is the package solution to the probabilistically unconstrained problem $Q^D(S)$. Note that, given a set $O$ of optimization scenarios, we can approximate $V_0$ as previously discussed:

$$\hat{V}_0 = \sum_{i=1}^{n} \widehat{\text{CVaR}}_\alpha(t_i.A, O) * x_i^D. \tag{4}$$

For any "non-trivial" risk constraint, i.e., a constraint that is not satisfied by $x^D$, setting the L-CVaR parameters to $(V', \alpha') = (V_0, 1)$ will necessarily yield a validation-infeasible package that does not satisfy the constraint. We thus need to search for the least restrictive value of $V'$ and $\alpha'$ between the limits $[V_0, V]$ $[\alpha, 1]$ respectively that results in a validation-feasible and $\epsilon$−optimal package solution.

**Alternating parameter search (APS).** The RCL process starts by setting the L-CVaR constraint values to $(V', \alpha') = (V, 1)$. Using the maximal value of $\alpha'$ will most likely make the package solution validation-infeasible. To attain feasibility, we make the L-CVaR constraint more restrictive via an *$\alpha$-search*, that is, by decreasing $\alpha'$, using bisection, down to the maximum value for which the resulting package is validation-feasible, while keeping $V'$ unchanged (line 13)—decreasing $\alpha'$ further would lower the objective value. After the above $\alpha$-search terminates with a validation-feasible package, we improve the objective value (while maintaining feasibility) by making the L-CVaR constraint less restrictive via a *$V$-search*, that is, by decreasing $V'$, via bisection, down to the maximum value for which the corresponding package remains validation-feasible, while keeping $\alpha'$ unchanged (line 18). Each bisection search for $V'$ terminates when the length of the search interval falls below a specified small constant $\delta$; in our experiments we found that $\delta = 10^{-3}$ was an effective and robust choice.

We then pivot back to adjusting $\alpha'$. Specifically, we decrease the new value of $V'$ by $\delta$, so that the corresponding package based on $(V' - \delta, \alpha')$ is now validation-infeasible. We then decrease $\alpha'$ using an $\alpha$-search to regain feasibility, then decrease $V'$ via a $V$-search to improve the objective, and so on. If any intermediate package is validation-feasible for the original SPQ $Q(S)$ and is near-optimal, i.e., the objective value equals or exceeds $(1 - \epsilon)\omega_0$, then we immediately terminate the RCL search and return this package as our solution, setting the variable Done to True (lines 15 and 20). The foregoing procedure works even if the initial $(V, 1)$ L-CVaR constraint yields a validation-feasible package; in this case the first

$\alpha$-search will trivially result in $(V', \alpha') = (V, 1)$, and the subsequent $V$-search will try to improve the objective value.

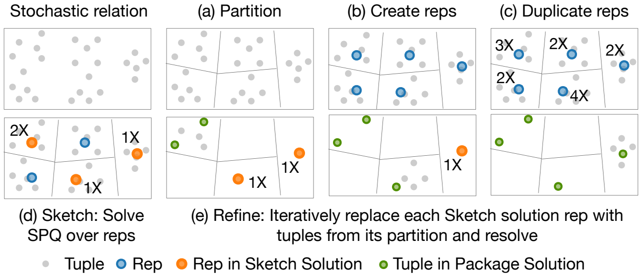
Our motivation for using APS is that it provably finds the optimal L-CVaR parameterization in the idealized setting where we can solve any SPQ exactly, determine the feasibility and objective value for any package exactly, and compute any VaR or CVaR exactly, without the need for SAA. Specifically, for an SPQ $Q(S)$ with one risk constraint parameterized by $(\alpha, V)$, denote by $\mathcal{F}_{Q(S)}$ the set of all L-CVaR parameterizations $(\alpha', V') \in [\alpha, 1] \times [V_0, V]$ for which the transformed SPQ $Q'(S; \alpha', V')$ with linearized risk constraint admits a feasible package solution. For any $(\alpha', V') \in \mathcal{F}_{Q(S)}$, denote by $\text{Obj}(\alpha', V')$ the objective value of its corresponding package solution. Intuitively, let $(\alpha^*, V^*)$ be the best parametrization in $\mathcal{F}_{Q(S)}$. Since APS minimally decreases $\alpha'$ such that the resulting package is validation-feasible (while maximizing the objective with the current value of $V'$), it is necessary that APS finds the point $(\alpha^*, \tilde{V})$ for some $\tilde{V}$. The following $V$-search will find $(\alpha^*, V^*)$. Formally, we have the following result:

Theorem 3.1. *Let $Q(S)$ be is a solvable SPQ with one risk constraint and suppose that $\mathcal{F}_{Q(S)}$ is nonempty. Then APS, under the idealized-setting assumption, will find a parameterization $(\alpha^*, V^*) \in \mathcal{F}_{Q(S)}$ such that $Obj(\alpha^*, V^*) \geq Obj(\alpha', V')$ for all $(\alpha', V') \in \mathcal{F}_{Q(S)}$.*

In practice, we run APS using a set $\mathcal{O}$ of optimization scenarios to compute packages and validation scenarios $\mathcal{V}$ assess feasibility and objective values. The ideal setting essentially corresponds to $|\mathcal{O}| = |\mathcal{V}| = \infty$, so we expect APS to work increasingly well as $|\mathcal{O}|$ and $|\mathcal{V}|$ increase. Our experiments in Section 6 indicate good practical performance for the ranges of $|\mathcal{O}|$ and $|\mathcal{V}|$ that we consider.

**Multiple risk constraints.** In general the set $R$ of risk constraints appearing in $Q$ satisfies $|R| > 1$, so that quantities such as $V$ and $\alpha$ are vectors; the function GetParams parses the query $Q$, extracts the values $V_r$ and $\alpha_r$ for each constraint $r \in R$, and organizes the values into the vectors $V$ and $\alpha$ (line 5). The QuickSolve$(Q, S)$ function works basically as described above. Assuming that qsSuccess = False, so that the full RCL procedure is needed, the linearization process starts by setting $\alpha'_r = 1$ and $V' = V_r$ for each $r \in R$, so that one or more of the risk constraints $r \in Q$ are validation-violated; denote by $R^* \subseteq R$ the set of problematic risk constraints. The search then decreases the problematic $\alpha'_r$ values via individual $\alpha$-searches until all the SPQ constraints are validation-satisfied. These $\alpha$-searches are synchronized in that, at each search step, a bisection operation is executed for each $\alpha'_r$ value with $r \in R^*$ and then the resulting overall set of risk constraints is checked for validation-satisfaction. The $\alpha$-search terminates when bisection has terminated for all of the problematic constraints due to convergence.[3] When the $\alpha$-search terminates, the RCL procedure next decreases each $V'_r$ value by $\delta$ and commences a set of synchronized $V$-searches, and the two types of search alternate as before until a solution is produced.

**Increasing the number of optimization scenarios.** We have been assuming that the initial number $m$ of scenarios is sufficiently large so that the SAA ILP well approximates the true SPQ, but this may not be the case in general. Therefore, right after each (synchronized) bisection step during an $\alpha$-search or $V$-search, the RCL process examines the intermediate package corresponding to the new set of constraints just produced. If there is a significant relative difference between any VaR, CVaR, or expected value when computed over the optimization scenarios and when computed over the validation scenarios, then it follows that SAA approximation is poor. If any relative difference is larger than $\epsilon$, then the $\alpha$-search or $V$-search terminates immediately and sets NeedScenarios to True (lines 14 and 19). We then increase the number of scenarios (line 23) and restart the RCL process. We also increase the number of scenarios if the values of both $\alpha'$ and $V'$ stop changing and a solution package has not yet been found (line 22). If the number of optimization scenarios starts to exceed the number of validation scenarios, we terminate the algorithm and return the best solution so far (line 23).



Stochastic relation

(a) Partition

(b) Create reps

(c) Duplicate reps

(d) Sketch: Solve SPQ over reps

(e) Refine: Iteratively replace each Sketch solution rep with tuples from its partition and resolve

**Figure 2: Solving a stochastic package query (SPQ) with Stochastic SketchRefine.**

**Termination and guarantees.** Assuming that the set $\mathcal{V}$ of validation scenarios is sufficiently large and that the upper bound $\omega_0$ is sufficiently tight, RCL-Solve will terminate at line 4, 15, or 20 with a near-optimal solution to $Q(S)$ or will terminate at line 4 with a NULL solution indicating that the problem is unsolvable. This is the behavior we observed in all of our experiments. Otherwise, RCL-Solve will terminate at line 24 and return the best package found so far, but with no guarantees on optimality. In this case, the user can increase the number of validation scenarios and try again.

## 4 STOCHASTIC SKETCHREFINE

RCL-Solve alone cannot handle SPQs with very large numbers of tuples. To achieve scalability across the tuple dimension, we propose a new evaluation mechanism, Stochastic SketchRefine. This algorithm follows the divide-and-conquer approach of SketchRefine [4] but makes the necessary (and nontrivial) modifications needed to handle uncertain data.

Figure 2 gives a high-level illustration of the processing steps. The algorithm first partitions the data offline into groups of similar tuples (Figure 2a) using DistPartition, a novel trivially-parallelizable stochastic partitioning scheme (Section 5) which ensures that (1) tuples in the same partition are similar, and (2) the maximum number $\tau$ of tuples allowed in each partition is small enough so that an ILP based on these tuples can be solved quickly.

Stochastic SketchRefine then creates a *representative* for each partition (Figure 2b). A representative of a partition is an artificial tuple in which the value of each deterministic attribute is the average value over the tuples in the partition, and the stochastic-attribute distributions equal those of a selected tuple from the partition; Section 4.1 below details the selection process.

[3] In theory, an initially non-problematic constraint might become problematic during the $V$-search, but we did not observe this behavior in any of our experiments. In any case, one can modify the algorithm slightly to deal with this rare situation.

**Algorithm 2** SolveSketch

```
Input: $Q$ := A Stochastic Package Query
       $T^{\mathrm{R}}$ := A set of representative tuples
       $\tau$ := Maximum number of tuples in a partition
       $\Delta\Gamma$ := Change in relative risk tolerance
       $m$ := Initial number of optimization scenarios
       $\mathcal{V}$ := Set of validation scenarios
       $P_{\max}$ := Max. number of distinct tuples in a package
       $\epsilon$ := Error approximation bound
       $\delta$ := Bisection termination distance
Output: $x$ := A validation-feasible + $\epsilon$-optimal package for $Q(S_0)$
 1: qsSuccess, $x^{\mathrm{D}} \leftarrow$ QuickSolve($Q, T^{\mathrm{R}}$)      ▷ Is the solution "easy"?
 2: if qsSuccess = True then
 3:     return $x^{\mathrm{D}}$                                     ▷ Either sketch solution or NULL
 4: $\Gamma \leftarrow$ InitialRiskTolerance($\tau$)
 5: while $m \leq |\mathcal{V}|$ do
 6:     $S_0 \leftarrow \emptyset$
 7:     for $t \in T^{\mathrm{R}}$ do
 8:         $d \leftarrow$ NumberOfDuplicates($t, Q, \Gamma, m, P_{\max}$)
 9:         $H_t \leftarrow$ Partition($t$)                       ▷ Set of tuples represented by $t$
10:         $\overline{\rho} \leftarrow$ MedianCorrelation($t, H_t$)
11:         $S_0 \leftarrow S_0 \cup$ GenDuplicates($t, d, \overline{\rho}$)
12:     $x_0$, NeedsScenarios $\leftarrow$ RCL-Solve-S($Q, S_0, m, \mathcal{V}, \delta, \epsilon$)
13:     if NeedsScenarios = True then
14:         $m \leftarrow \min(2m, |\mathcal{V}|)$
15:     else
16:         if $x_0 \neq$ NULL then return $x_0$
17:         else
18:             if $\Gamma > 0.0$ then
19:                 $\Gamma \leftarrow \max(\Gamma - \Delta\Gamma, 0.0)$
20:             else
21:                 $m \leftarrow \min(2m, |\mathcal{V}|)$
22: return NULL                                    ▷ means the problem is unsolvable
```

The next step is to create a *sketch* $S_0$ using the representatives. In the deterministic setting, it suffices to simply define the sketch as the union of the representatives; the corresponding package solution specifies a multiplicity $x(t) \in [0, 1, 2, \ldots]$ for each representative $t$. In the stochastic setting, however, this can lead to infeasible or sub-optimal packages due to inflation of risk—see Section 4.1 below—so instead of relying solely on multiplicities, we augment the sketch $S_0$ with distinct but stochastically identical *duplicates* of each representative (Figure 2c). To account for statistical correlation between the tuples within a partition, the duplicates must also be mutually correlated—a Gaussian copula method called NORTA is used to generate correlated duplicates [17].

Stochastic SketchRefine then solves $Q(S_0)$ using a slight variant of RCL-Solve, called RCL-Solve-S, thereby assigning a multiplicity to each duplicate (Figure 2d) and creating a package solution. This package solution is then *refined* into a new set of tuples $S_1$ by selecting a (subset of) partitions containing at least one tuple appearing in the package solution and replacing the duplicates in these partitions by the original tuples in them (Figure 2e); because the size of each partition is bounded, the resulting query $Q(S_1)$ can be solved quickly using a slight variant of RCL-Solve called RCL-Solve-R. This refinement process continues until all duplicates in the sketch solution are replaced by tuples from their respective partitions, thus obtaining the final package solution. The following sections describe these algorithmic components in detail.

## 4.1 The Sketch Phase

SolveSketch (Algorithm 2) creates a sketch $S_0$ using the set $T^{\mathrm{R}}$ of representatives produced by DistPartition and then solves the resulting query $Q(S_0)$ to create an initial sketch solution package $x_0$. As with RCL-Solve, the QuickSolve function tries to solve the deterministic package query $Q^{\mathrm{D}}(T^{\mathrm{R}})$ obtained by removing all risk constraints from $Q$. If a solution $x^{\mathrm{D}}$ exists that is validation-feasible with respect to $Q$, it is returned as the sketch solution $x_0$; if a solution to $Q^{\mathrm{D}}(T^{\mathrm{R}})$ does not exist, then SolveSketch declares the sketch problem unsolvable and returns NULL (lines 1–3).

If qsSuccess = False, then the full query $Q$ must be solved. The key difference from the deterministic setting is the need to create duplicates of each representative in $T^{\mathrm{R}}$ to form the initial sketch $S_0$. The key issues are (i) determining the number of duplicates to create for each representative, (ii) determining the appropriate correlation between a set of duplicates, and (iii) generating scenarios from correlated duplicates. We first motivate the use of duplicates and then discuss how issues (i) and (ii) are addressed; see [23, Appendix C] for details on how the NORTA procedure is used to generate scenarios from correlated duplicates.

**The need for duplicates.** Duplicates are used in a partition $P$ to ensure that optimal tuples in $P$ do not get erroneously eliminated from consideration due to overestimation of risk. As a simple example, consider a partition containing tuples $t_1, \ldots, t_k$ having a stochastic attribute $A$ such that $t_i.A$ has an independent $N(0,1)$ distribution (normal with mean 0 and variance 1) for all $t_i \in P$. Also suppose that the SPQ has a constraint of the form `sum(A)<= -2.0 with probability <= 0.1`. Suppose that the truly optimal package contains two tuples $t_1$ and $t_2$ from $P$, each with a multiplicity of 1. The sum $t_1.A + t_2.A$ has a $N(0,2)$ distribution, so that $\mathbb{P}(t_1.A + t_2.A \leq -2.0) \approx 0.08$ and there are no feasibility issues. Here, losses from one tuple can be offset by gains in the other tuple, reducing the overall risk of high losses. However, if there is only a single representative $t_p \in P$ in the sketch of $P$, then a candidate sketch package that tried to use two tuples from $P$ (as in the optimal package) would end up using $t_p$ with multiplicity 2, so that $2t_p.A \sim N(0,4)$ and $\mathbb{P}(2t_p.A \leq -2.0) \approx 0.16$, violating the constraint. Due to this overestimation of risk, the representative $t_p$ can have a multiplicity of at most 1 in the sketch package. Suppose that another representative $t_r$ from a different partition is also in the sketch package (a typical outcome). Then we cannot refine $[t_p : 1, t_r : m]$ to the intermediate package $[t_1 : 1, t_2 : 1, t_r : m]$ for any $m \geq 1$ since this package is necessarily infeasible; if it were feasible, then $[t_1 : 1, t_2 : 1]$ would not be optimal since the objective of $[t_1 : 1, t_2 : 1, t_r : m]$ would be greater. Thus, the final package would have at most one of $t_1$ and $t_2$, and hence be suboptimal. We solve this problem by maintaining duplicates $t_p^{(1)}$ and $t_p^{(2)}$. Then, as with the actual tuples, a sketch solution could contain these duplicates, and not violate the constraint. After $P$ is refined, the solver would have the option of using both tuples $t_1$ and $t_2$ in the final package solution. In general, tuples within a partition might be correlated; in this case, the duplicates would also be correlated to accurately approximate the risk of using tuples in the partition when forming a package. We emphasize that, because of the refinement step, actual tuples, and not duplicates, appear in the final package solution. We provide an ablation study that empirically demonstrates the deleterious effect of not using duplicates [23, Appendix I.1].

**Choosing the number of duplicates.** We assume that we have an upper bound $P_{\max}$ on the number of distinct tuples in a package. This can often be derived from query constraints such as

COUNT(*) $\le P_{\max}$, or SUM(A) $\le V$, or on domain knowledge. In an extreme case, all of the $P_{\max}$ tuples in the optimal package might belong to the same partition; if each representative has $P_{\max}$ duplicates and if the optimal tuples are close to the representative, the sketch will yield a validation-feasible package. However, since each duplicate induces a corresponding decision variable, solver runtimes can be expensive.

We therefore aim to use $d_t < P_{\max}$ duplicates for each representative $t$. Doing so incurs some risk: if the optimal package contains $P_{\max}$ tuples and all of these tuples come from $t$'s partition, then the average multiplicity of each duplicate in the sketch solution is $P_{\max}/d_t$. As we have seen, assigning a multiplicity $\geq 2$ to any duplicate results in an increased probability of each risk constraint $r \in R$ being violated. Specifically, for a given constraint $r$ that imposes a lower bound on a risk metric (VaR or CVaR), the risk value of the package based on $d_t$ duplicates in the sketch will be lower than that of the package formed using $P_{\max}$ duplicates and hence more likely to violate the lower bound. We measure the relative increase in risk with respect to $r$ due to using $d_t$ duplicates instead of $P_{\max}$ duplicates by $\gamma(r, d_t) = \big(\text{Risk}_r(S_{\max}) - \text{Risk}_r(S_{d_t})\big)/|\text{Risk}_r(S_{\max})|$, where $S_{\max}$ and $S_{d_t}$ are sketches consisting of $P_{\max}$ and $d_t$ duplicates of $t$, and $\text{Risk}_r$ is the VaR or CVaR function that appears in constraint $r$. Note that $\gamma(r, d)$ increases with decreasing $d$. We determine the number of duplicates $d_t(r)$ as the smallest $d$ such that $\gamma(r, d) \le \Gamma$, where $\Gamma$ is a specified *risk tolerance ratio.* We then set $d_t = \min(\max_{r \in R} d_t(r), |P_t|)$, where $|P_t|$ is the number of tuples in the $t$'s partition. (Thus the number of duplicates is upper-bounded by the number of tuples in $P_t$.) The function NumberOfDuplicates in line 8 performs these calculations.

Note that, as $\Gamma$ decreases, the number of required duplicates $d$ increases. Using bisection, we initially choose $\Gamma \in [0, 1]$ such that $\sum_{t \in T^{\mathrm{R}}} d_t \le \max(\tau, |T^{\mathrm{R}}|)$, where $T^{\mathrm{R}}$ is the set of representatives and $\tau$ is an upper bound on the total number of duplicates (line 4). The bound $\tau$ is set such that in-memory ILP solvers like Gurobi can solve problems with $\tau$ tuples within an acceptable amount of time. The number of tolerable decision variables can vary due to underlying hardware attributes of the system, differences in the solver software, runtime versus quality requirements, and so on.

**Choosing the correlation between duplicates.** We have defined duplicates to be distinct but stochastically identical, by which we mean that for a set $t^{(1)}, \ldots, t^{(d)}$ of duplicates and each stochastic attribute $A$, the marginal distributions of $t^{(1)}.A, \ldots, t^{(d)}.A$ are the same. To motivate the notion of duplicates we gave a small example using two statistically independent duplicates having a common $N(0, 1)$ marginal distribution. In general, however, assuming mutual independence among duplicates ignores correlations between the actual tuples within a partition, which can cause problems in the refine phase. E.g., if all tuples in the partition are highly correlated, then sums of the form $\sum_i t_i.A * x_i$ are subject to more severe fluctuations than sums involving independent $t_i.A$'s, and the risk of violating VaR or CVaR constraints is higher. Thus, a sketch package with independent duplicates for a given partition may underestimate the risks of including actual tuples from that partition, and no feasible solution may be found during refine, when the solver tries to replace independent duplicates with positively-correlated tuples. Duplicates of each representative should therefore roughly mirror the correlation between tuples in the partition.

We use the median $\overline{\rho}$ of the pairwise Pearson correlation coefficients between the representative and all other tuples in its partition as the correlation coefficient between each pair of duplicates (line 10). Using the median ensures that the number of tuples having a higher correlation with the representative relative to the duplicates equals the number of tuples having a lower correlation with the representative. Thus, if one of the duplicates is replaced by a tuple having a higher correlation coefficient than the median, thereby increasing risks compared to the sketch package, the solver will have the chance to balance it out by taking another similarly optimal tuple that has a lower correlation coefficient. (We actually take the maximum of $\overline{\rho}$ and 0 so that the correlation matrix for the duplicates is positive semi-definite. Our clustering method—see Section 5— ensures that that the median value is virtually always nonnegative so that no correction is needed.)

**Computing the sketch package.** As discussed, SolveSketch initially sets $\gamma$ in line 4 so as to ensure at most $\tau$ duplicates overall. In lines 7–11 the initial set of duplicates are created to form an initial version of the sketch $S_0$. Then a slight variant of RCL-Solve is used to try and solve $Q(S_0)$ (line 12). This variant, RCL-Solve-S, is almost identical to RCL-Solve, except that the process of increasing the number of scenarios is now controlled by the calling function SolveSketch. Specifically, if NeedScenarios = True as in lines 14 or 19 of RCL-Solve, then a NeedScenarios indicator variable is set to True and returned to SolveSketch. Moreover, lines 23–25 in RCL-Solve are replaced by a simple "**return** NULL" statement. Thus if the need for more scenarios is detected during an $\alpha$-search or $V$-search within RCL-Solve-S, this information is immediately sent to SolveSketch, which then increases the number of scenarios (lines 13 and 14). If RCL-Solve-S returns NULL, i.e., if the number of optimization scenarios appears adequate but an $\epsilon$-optimal and validation-feasible solution cannot be found while attempting to solve the sketch problem, then $\Gamma$ is decreased (line 19) and the resulting, larger sketch is tried. If $\Gamma = 0$ so that a decrease is impossible, we double the number of optimization scenarios (line 21) and try again, keeping $\Gamma = 0$ since we know that we will need a lot of duplicates. Note that we try decreasing $\Gamma$—thereby adding duplicates—before we increase the number of scenarios because generating more optimization scenarios can incur significant computational costs, and is hence deferred as long as possible.

## 4.2 The Refine Phase

The refinement process is very similar to that of the deterministic SketchRefine algorithm, so we give a brief overview and refer to reader to [4] for further details. The process iteratively replaces each synthetic representative selected in the sketch package with actual tuples from its own partition. First, using the 'Best Fit Decreasing' algorithm for bin-packing [15], the Refine algorithm bins the partitions with tuples in the sketch package into a near-minimum number of 'partition groups' such that total number of tuples in each partition group is at most $\tau$. Then, in each step, it replaces all the selected duplicates from one of the partition groups, while preserving both the selected duplicates from the as yet unrefined partitions and the tuples selected during previous refinements.

**Number of optimization scenarios.** Refine uses as many optimization scenarios as were used to derive the sketch package. The intuition is that if $m$ optimization scenarios were enough to create a satisfactory package from the representative duplicates, they should suffice for doing the same from the actual tuples, since each tuple should be similar to their representatives.

**Refinement order.** Starting from a sketch package with tuples in $k$ distinct partitions that are binned into $b$ groups, there are $b!$ possible orders in which the partitions can be refined. Using an 'incorrect' order can lead to an infeasible intermediate ILP, or one whose package solution is far less optimal than the sketch package. Refine attempts to select a correct order using a "greedy backtracking" technique as in [4]. The idea is to start with a randomly selected permutation of partition groups, i.e., refinement order. If an intermediate refinement fails to find a validation-feasible package whose objective value is within $(1 \pm \epsilon)$ of the objective value of the sketch package, we "undo" the previous refinement and greedily attempt to refine the failing partition group in its place. We keep undoing the previous refinements until the partition group can be successfully refined. If the group cannot be refined even when it is moved back to the first position in the refinement order, then the stochastic behaviour of the tuples within the partition group is not adequately captured by the behavior of the duplicates for the partition. This can happen if the correlations among the duplicates are too small (Section 4.1), so we increase the common correlation coefficient for each of those partitions from $\overline{\rho}$ to $\overline{\rho} + \Delta_\rho$ (we take $\Delta_\rho = 0.1$). This discourages the sketch solver from taking more tuples from these partition groups. Afterwards, we re-execute the sketch query, and refine the newly obtained sketch package. Greedy backtracking continues to explore different orders in which the groups may be refined until an acceptable final package is found.

**Refinement operation.** At each step, Refine selects an unrefined partition group with at least one duplicate appearing in the sketch package. The refinement operation involves solving an ILP corresponding to $Q(S_c \cup S_p \cup S_u)$, where $Q$ is the SPQ of interest, $S_c$ comprises all of the actual tuples for the partition group currently being refined, $S_p$ comprises package tuples remaining from previously-refined groups, and $S_u$ comprises the duplicates from all as yet unrefined groups. The constraints of the ILP include all constraints in $Q$ (including the linearized risk constraints), as well as additional constraints ensuring that the multiplicities of the tuples in $S_p$ and $S_u$ remain unchanged. Thus the only change to the package is to replace the duplicates for the current group with zero or more actual tuples from its partitions. We use the variant RCL-Solve-R to formulate and solve the ILP. The only differences between RCL-Solve and RCL-Solve-R are that (i) in line 6 the upper-bound constant $\omega_0$ is instead chosen as the objective value $\bar{\omega}$ corresponding to the package solution of the sketch problem $Q(S_0)$ produced by SolveSketch, (ii) in line 23, NULL is returned rather than the best feasible solution so far and (iii) the number of optimization scenarios is not increased. Thus a non-NULL package returned by RCL-Solve-R will be validation-feasible with an objective value $\omega$ that satisfies $\omega \geq (1 - \epsilon)\bar{\omega}$.

# 5 STOCHASTIC PARTITIONING

Stochastic SketchRefine needs tuples in a relation to be partitioned into sufficiently small groups of similar tuples prior to query execution. Specifically, each refine ILP has at least as many decision variables as tuples in the partition being refined. We therefore constrain the number of tuples in every partition to a given size threshold $\tau$. The parameter $\tau$ is as in Algorithm 2 and is chosen to ensure reasonable ILP-solver solution times. Also, Stochastic SketchRefine requires high inter-tuple similarity within partitions. During sketch, every tuple in a partition is represented by a single representative, and a representative that is not sufficiently similar to all the tuples in its partition may result in sketch packages that cannot be refined to feasible and near-optimal packages. Finding a suitable representative can be hard if tuples within a partition are too dissimilar with respect to their attributes.

**Prior Clustering Methods.** Existing uncertain data clustering algorithms that cluster similar stochastic tuples together do not ensure the resulting clusters will satisfy size thresholds [13, 18, 19, 22]. Although some hierarchical partitioning approaches [20, 40] can be modified to repeatedly repartition the clusters until no cluster has more than $\tau$ tuples, their runtime complexities are super-quadratic with respect to the number of tuples in the relation, making them unsuitable for fast partitioning of large relations. We therefore introduce DistPartition, a partitioning algorithm having sub-quadratic time complexity that ensures no partition in a stochastic relation has more than $\tau$ tuples. We provide a high-level overview of the working principles of DistPartition here, and refer interested readers to [23, Appendix D.3] for more details.

**Diameter Thresholds.** To ensure that tuples within a partition are sufficiently similar, DistPartition ensures that the "distance" (as defined below) between any pair of tuples in a partition with respect to each attribute $A$ is upper-bounded by a *diameter threshold* $d_A$. Values of $d_A$ for every attribute $A$ can be chosen by the user based on runtime requirements. Smaller values of $d_A$ produce tighter partitions whose representatives better reflect the stochastic properties and deterministic values of other tuples in their partitions, leading to better quality solutions. However, this increases the number of partitions, thereby increasing the number of decision variables in the sketch ILP and consequently increasing runtime. In our experiments, we chose values of $d_A$ that keep the total number of partitions to approximately within $[\frac{\tau}{10}, \frac{\tau}{2}]$, to allow some wiggle room for Sketch to create duplicates without exceeding the size threshold $\tau$ or incurring excessive runtimes.

**Inter-tuple distances.** We calculate the distance between any two tuples $t_1$ and $t_2$ w.r.t. an attribute $A$ using their Mean Absolute Distance (MAD), which we define as $\text{MAD}(t_1.A, t_2.A) = \mathbb{E}[|t_1.A - t_2.A|]$. Thus, if $A$ is deterministic, then $\text{MAD}(t_1.A, t_2.A) = |t_1.A - t_2.A|$; if $A$ is stochastic, then we estimate MAD as the average of $|t_1.A - t_2.A|$ over a set of i.i.d. scenarios. We assume throughout that $\mathbb{E}[|t.A|] < \infty$ for all tuples $t$ and attributes $A$, so that the MAD is always well defined and finite. Unlike other metrics such as Wasserstein distance [31] and KL-divergence [35], which only consider similarities between probability density functions, MAD inherently accounts for correlations between tuples. Given $t_1$ and $t_2$ with similar PDFs for attribute $A$, the distance $\text{MAD}(t_1.A, t_2.A)$ is smaller when $t_1$ and $t_2$ are positively correlated than when they are independent;

the case study in [23, Appendix D.1] gives a concrete example of this phenomenon. If inter-tuple MADs within a partition are small, then the tuples in the partition will have similar values across scenarios, allowing a representative to closely reflect their stochastic behavior. This property is formally established via Theorem 5.1, which shows that bounding the MAD between two tuples also bounds the difference in their CVaRs at the tails of their distributions.

Theorem 5.1. *Suppose the MAD between tuples $t_1$ and $t_2$ w.r.t. an attribute $C$ is bounded by $d_C$, i.e., $\mathbb{E}[|t_1.C - t_2.C|] \le d_C$. Then, the difference between the CVaRs of $t_1$ and $t_2$ in their lower $\alpha$-tail is bounded: $|CVaR_\alpha(t_1.C) - CVaR_\alpha(t_2.C)| \le \frac{d_C}{\alpha}$*

We give a formal proof in [23, Appendix B]. Intuitively, consider what happens if the theorem's conclusion is false in an SAA setting: If the average value of $|t_1.C - t_2.C|$ exceeds $\frac{d_C}{\alpha}$ in any set of $\lfloor \alpha m \rfloor$ lowest-valued scenarios, then even if there is no (absolute) difference between their values in the remaining scenarios, $\mathbb{E}[|t_1.C - t_2.C|] > \alpha \frac{d_C}{\alpha} = d_C$, thus contradicting the hypothesis. For efficiency, DistPartition generates a fixed set of scenarios before partitioning to avoid repeatedly generating scenarios on the fly. Using more scenarios makes MAD estimation more accurate for stochastic attributes. In our experiments, we precomputed a set of 200 scenarios. See [23, Appendix D.2] for insight on how many scenarios are needed to obtain statistically accurate estimates of MAD. Using MAD allows us to formally guarantee that Stochastic SketchRefine achieves a $(1 - \epsilon)^2$-optimal solution; see [23, Appendix E].

**Triggering the partitioning of a set.** DistPartition recursively partitions sets of stochastic tuples until no size or diameter constraints are violated. The size constraint is violated if a set has more than $\tau$ tuples. Exactly determining the diameter of a partition for an attribute $A$ would require estimating the MAD between every pair of tuples in the set. To avoid this quadratic complexity, we use a conservative approach which exploits the fact that MAD, because it is based on the $L_1$ norm, inherits the triangle inequality: $\text{MAD}(t_1.A, t_3.A) \le \text{MAD}(t_1.A, t_2.A) + \text{MAD}(t_2.A, t_3.A)$. In an operation we call PivotScan, we find the distance of every other tuple in the set from a randomly chosen tuple $t$. Hence, if the distance of the farthest tuple $\dot{t}_A$ from $t$ is $\dot{d}_A$, the distance between any two tuples in the partition is bounded by $2\dot{d}_A$. If $2\dot{d}_A \le d_A$ is true for each attribute $A$, all the diameter constraints are necessarily satisfied. If any constraint is violated, further partitioning is triggered.

**PivotScan-based partitioning.** To partition further, DistPartition (1) executes a PivotScan for each attribute from a random tuple $t$ to identify the attribute $A^*$ with the highest diameter-to-threshold ratio: $A^* = \arg\max_A 2\dot{d}_A/d_A$, (2) runs a second scan over all tuples in the set, but this time calculates their distances from the farthest tuple $\dot{t}_{A^*}$ found in the first scan, and (3) stores a list of tuple IDs in increasing order of distance from $\dot{t}_{A^*}$. Further partitioning is done by segmenting the list and creating a sub-partition out of each segment. Specifically, if the size constraint is violated, then we partition the list into contiguous segments each containing $\le \tau$ tuples. Each resulting segment thus satisfies the size constraint. If a diameter constraint is violated by any resulting segment (according to the conservative test described previously), it is recursively partitioned using distance-based partitioning. Let $\ddot{d}_{A^*}$ be the distance of the farthest tuple in the sub-partition from $\dot{t}_{A^*}$. For distance-based partitioning, we create $\lceil \ddot{d}_{A^*}/d_{A^*} \rceil$ sub-partitions where the $i$-th sub-partition contains all tuples within distance $[(i-1) \cdot d_{A^*}, i \cdot d_{A^*}]$. Each sub-partition is then recursively partitioned until no diameter constraints are violated. PivotScan is embarassingly parallel, and, after the initial size-based partitioning, diameter-based partitioning can be conducted in parallel for each partition. DistPartition can hence be accelerated with multi-core processing.

**Representative Selection.** After partitioning, we select a representative tuple for each partition. The value of each deterministic attribute of the representative tuple is equal to the mean of that attribute among every tuple in the partition. For stochastic attributes, naively computing a "mean distribution" is computationally expensive. Thus the distribution of each stochastic attribute of the representative is made equal to that of a chosen tuple in the partition, preferably the tuple with the lowest mean distance to every other tuple. However, finding the tuple with the lowest mean distance requires quadratic MAD estimations. Instead, we propose the total *worst-case replacement* cost heuristic for selecting representatives. First, we compute the minimum and maximum values of an attribute $A$ over all the tuples in each scenario. In a given scenario, we define the worst-case *replacement cost* of $t.A$ for a tuple $t$ as the greater of its absolute difference with that scenario's minimum or maximum. The cost for $t.A$ is obtained by summing its replacement costs over all the scenarios. The representative variable for $A$ is then chosen as that of the tuple with the minimum total cost. We provide pseudocodes for the DistPartition and representative selection routines in [23, Appendix D.3] and [23, Appendix D.4].

This representative tuple selection scheme assumes that values for different stochastic attributes are generated independently. See [23, Appendix D.5] for a description of the minor modifications to representative selection and scenario generation when the stochastic attributes within a tuple can be statistically correlated.

## 6 EXPERIMENTAL EVALUATION

We show that (1) RCL-Solve produces packages of comparable quality as SummarySearch in an order of magnitude lower runtime and (2) Stochastic SketchRefine scales to significantly larger data sizes than both of these approaches. Additional experiments in [23, Appendix I] justify the use of duplicates and DistPartition.

### 6.1 Experimental Setup

**Environment.** All approaches are implemented in Python 3.12.14. We use Postgres 16.1 as the supporting DBMS, and Gurobi 11.0.3 as the base solver. We ran our experiments on a 2.66 GHz 16-core processor with 16 GB of RAM. For each experiment, we report average results and standard deviation error bars over 16 runs.

**Datasets.** We use two datasets in our experiments: (1) We construct a stock investments table (Figure 1) using historical NASDAQ, NYSE, and S&P-500 data [30] with up to 4.8M tuples. We model gains using Geometric Brownian Motion with drift and volatility estimated from historical stock prices for 3289 companies. We vary holding periods from a half a day to 730 days. (2) We generate up to 6M LineItem tuples from the TPC-H V3 benchmark [32]. We add Gaussian noise ($\mu_{\text{noise}} \sim \mathcal{N}(0, 1)$, $\sigma^2_{\text{noise}} \sim \text{Exp}(2)$) to price and quantity to introduce stochasticity. Further details are in [23, Appendix H.1].

**Workloads.** We generate a workload of SPQs following the methodology of [28]: We vary the constraint bounds of the SPQs to generate queries of varying *hardness* (H), measured as the negative log likelihood that a random package satisfies the query's constraints; see [23, Appendix H.2]. For stocks, the queries find portfolios that maximize expected total gains, such that total price is below a fixed budget and total gains exceed a threshold with a given high probability. For TPC-H, the queries find packages of items that maximize expected total prices, such that total price exceeds a minimum bound with a given high probability, total quantity shipped is below a limit with a given high probability, and total taxes are below a fixed amount. All packages have a size constraint of at most 30 tuples. We provide complete workload details in [23, Appendix H.2].

**Hyperparameters.** We set the number of initial optimization ($m$) and validation scenarios ($\hat{m}$) to 100 and $10^6$, respectively, for both SummarySearch and RCL-Solve. We use $\epsilon = 0.05$ as the approximation error bound, and set $\delta = 10^{-2}$ as the bisection termination threshold for RCL-Solve. For Stochastic SketchRefine, we set $P_{\max}$ to 30, and during Sketch, we decrease the relative risk tolerance by 0.03 in each iteration, i.e., $\Delta\Gamma = 0.03$. We set the partitioning size threshold $\tau = 10^5$, as Gurobi solves randomly-generated, satisfiable ILPs with 3 constraints in roughly one minute at this scale. We set the diameter thresholds for the Portfolio dataset as $d_{\text{price}} = 10$, $d_{\text{gain}} = 100$, and for TPC-H as $d_{\text{price}} = 50$, $d_{\text{quantity}} = 5$, $d_{\text{tax}} = 0.05$. [23, Appendix G] contains details on how our results are not sensitive to small changes in parameter values and provides guidance on finding appropriate hyperparameter settings for different datasets.

**Metrics.** We report wall-clock *query run times* and the *relative integrality gap* $(\omega - \omega^*)/\omega^*$, where $\omega$ is objective value of the returned package and $\omega^*$ is that of the best package found on a relaxation of the query with integrality constraints removed. The latter "best" package is one found by either RCL-Solve, SummarySearch, or (on data sets with fewer than 40K tuples) naïve [7]. We report the means ($\mu$) and standard deviations ($\sigma$) of the relative integrality gaps of packages formed by Stochastic SketchRefine (on larger relations) and RCL-Solve (on data sets with fewer than 40k tuples).

## 6.2 Main Results: Scalability and Optimality

**Increasing uncertainty.** We evaluated our novel RCL-Solve method against the state of the art in stochastic package query evaluation, SummarySearch. As we noted in Section 1, SummarySearch does not work well with high-variance stochastic attributes. In Figure 3, we demonstrate the performance of the two methods as the uncertainty in the stochastic attributes increases, while keeping the datasets small (40K tuples for Portfolio and 20K for TPC-H). We control uncertainty by modifying the volatility of the GBM model of a stock's gain, and the variance of Gaussian noise added to price and quantity in TPC-H. Increasing uncertainty also increases query hardness, as indicated by the hardness ranges reported in each plot. RCL-Solve is significantly faster than SummarySearch and this difference is more pronounced with high uncertainty and increased hardness. The reason is that RCL-Solve generates fewer scenarios than SummarySearch (approximately 6x fewer scenarios on average for the highest variance / volatility settings). Moreover, with many scenarios, SummarySearch not only solves more, but also harder optimization problems, that Gurobi takes minutes to solve.

The runtime gains of RCL-Solve do not come at the cost of quality. The packages it produced have a relative integrality gap in the 0.94–0.99 range, same as SummarySearch.

*Key takeaway:* Our novel risk linearization approach is superior to the state of the art, by maintaining robust quality and fast performance in cases of increased data uncertainty.

**Increasing data size.** While we saw RCL-Solve outperform SummarySearch, it alone cannot scale to larger data sizes. Stochastic SketchRefine provides the evaluation mechanism to achieve this scaling, while using RCL-Solve as a building block. In the experiment of Figure 4, we evaluate our query workload with all three methods, while increasing the data size to several million tuples. While RCL-Solve continues to outperform SummarySearch, their runtimes increase sharply when the data size is higher than $\tau$, and both methods fail to produce results on data larger than 1M tuples (or less in some cases), due to time-outs (runtime exceeds 1.5 hours) or crashes (Gurobi runs out of memory). In contrast, Stochastic SketchRefine maintains remarkably stable performance, thanks to its divide-and-conquer approach, ensuring that ILP subproblems fit within the available memory. On relations with size less than $\tau$, Stochastic SketchRefine and RCL-Solve have identical performance as Stochastic SketchRefine makes a single RCL-Solve call with the entire relation. The runtime benefits do not impact quality, and Stochastic SketchRefine produces packages that approximate the continuous relation of each SPQ with ratios ranging from 0.92 to 0.97. For the cases that SummarySearch and RCL-Solve produced results, those were also of high quality, with approximation ratios in the 0.95–0.98 range.

Query run times are higher for the Portfolio workload than TPC-H due to the expensive nature of sampling from a Geometric Brownian Motion model than from a Guassian one. Even so, since Stochastic SketchRefine requires fewer optimization scenarios and does not generate scenarios from all tuples at once, its runtime on the portfolio datasets stays under 20 minutes for all queries.[4]

*Key takeaway:* Stochastic SketchRefine remarkably scales to very large data, significantly outperforming the prior art.

# 7 RELATED WORK

**Probabilistic Databases** [9, 14, 26] specialize in representing data with stochastic attributes. Prior work has focused primarily on supporting SQL-type queries over uncertain data, allowing for ad hoc what-if analyses, as opposed to the in-database optimization that we support, which systematically searches through the space of possible decisions to find the optimal course of action.

**In-database decision-making** pushes decision making-related activities closer to where the data resides, and thus simplifies decision-making workflows and reduces unnecessary data movement. The work in [9] supports what-if analysis over historical data, but does not support full-scale optimization. SolveDB [36] and its successor, SolveDB+ [37] integrate a variety of black-box optimizers and their functionalities into a DBMS. However, they do not scale to large problems; presumably some package query solving techniques could be incorporated into SolveDB+. Recently,

[4] While validation scenarios are larger in number, they only need to be generated for few tuples, so they do not significantly impact runtime.

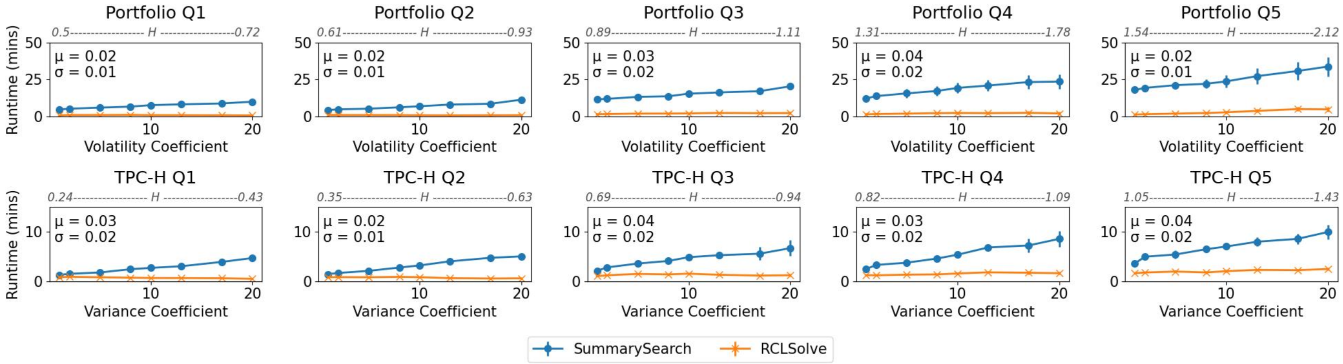


**Figure 3: Increasing variance or volatility coefficients increases tuple uncertainty as well as query hardness. ($H$ range reported with each plot). RCL-Solve is faster than SummarySearch especially at high variances. In each plot, $\mu$ and $\sigma$ report the mean and standard deviation, respectively, of the relative integrality gap for RCL-Solve's packages.**

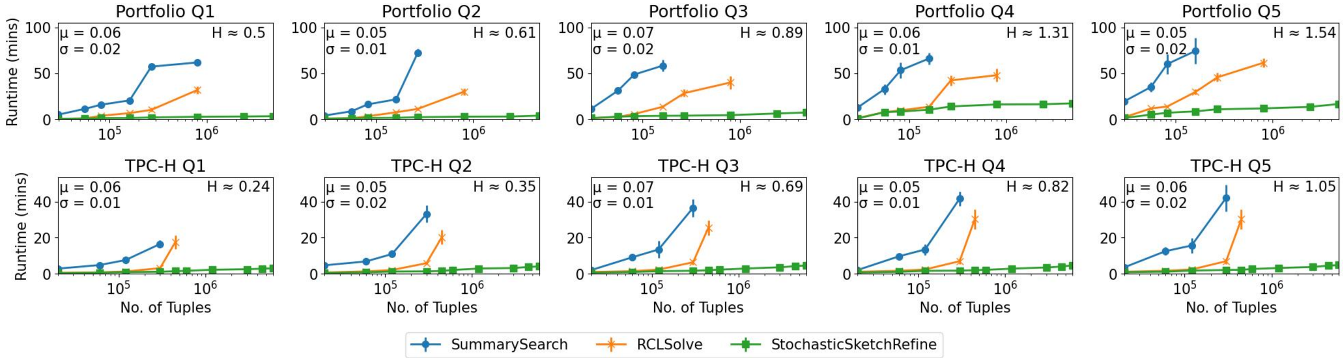


**Figure 4: RCL-Solve continues to outperform SummarySearch but fails to scale beyond 1M tuples. The absence of data points indicates that no solutions were found within 1.5 hours. Stochastic SketchRefine scales well as data size and query hardness ($H$) increase. Each plot shows the relative integrality gap statistics ($\mu$, $\sigma$) for Stochastic SketchRefine's packages.**

Progressive Shading [28] was able to improve over SketchRefine using a novel hierarchical partitioning scheme together with customized ILP solvers. An interesting direction for future work would be to extend this approach to stochastic data.

**Stochastic Optimization** has often used Monte Carlo sampling to estimate uncertain attributes that cannot be quantified exactly [24]. Constrained optimization involving probabilistic constraints can be hard to handle, as they can make the feasible regions non-convex [2]. Sample Average Approximation generates many scenarios to quantify these constraints using sample values [27]. Prior methods have often attempted to address this difficulty by reducing the number of scenarios [6, 10]. SummarySearch [6] uses conservative summaries to restrict the feasible region and improve feasibility. However, the feasible regions for the optimization problems remain nonconvex in general, and the search for an optimal set of summaries can be time consuming. Other methods have approximated non-convex risk constraints using their convex CVaR counterparts, but these approaches incur a quadratic time complexity for doing so [8]. RCL-Solve effectively summarizes all of the information contained in numerous scenarios with one linear constraint in the ILP, and finds the best set of L-CVaR constraints within a logarithmic number of iterations using bisection.

## 8 CONCLUSION AND FUTURE DIRECTIONS

In this work, we propose RCL-Solve, an improved SPQ solver that converts non-convex stochastic constrained optimization problems into ILPs whose size is independent of the number of scenarios. With Stochastic SketchRefine, we tackle scalability challenges in constrained optimization caused by the number of tuples in large relations. We further introduce DistPartition, an efficient approach for partitioning probabilistic relations. Together, these novel methods allow users, in the face of uncertainty and large datasets, to quickly compute near-optimal packages that satisfy a given set of constraints. In the future we wish to explore (i) how to relax stochastic query constraints to recommend alternative packages with better objectives that may slightly violate current constraints, (ii) how to scalably support sequential decision-making or two-stage stochastic programs, and (iii) how to explain SPQ solutions allowing for increased user trust with uncertain data.

## 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] Hervé Abdi and Lynne J Williams. 2010. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics* 2, 4 (2010), 433–459.

[2] Shabbir Ahmed and Alexander Shapiro. 2008. Solving chance-constrained stochastic programs via sampling and integer programming. In *State-of-the-Art Decision-making Tools in the Information-intensive Age.* (INFORMS), 261–269.

[3] Gordon J Alexander and Alexandre M Baptista. 2004. A comparison of VaR and CVaR constraints on portfolio selection with the mean-variance model. *Management science* 50, 9 (2004), 1261–1273.

[4] Matteo Brucato, Azza Abouzied, and Alexandra Meliou. 2018. Package queries: efficient and scalable computation of high-order constraints. *The VLDB Journal* 27 (2018), 693–718.

[5] Matteo Brucato, Juan Felipe Beltran, Azza Abouzied, and Alexandra Meliou. 2015. Scalable package queries in relational database systems. *arXiv preprint arXiv:1512.03564* (2015).

[6] Matteo Brucato, Nishant Yadav, Azza Abouzied, Peter J. Haas, and Alexandra Meliou. 2020. Stochastic Package Queries in Probabilistic Databases. In *Proc. ACM SIGMOD*. 269–283. https://doi.org/10.1145/3318464.3389765

[7] Matteo Brucato, Nishant Yadav, Azza Abouzied, Peter J. Haas, and Alexandra Meliou. 2021. Scalable package queries in relational database systems: Extended version. *arXiv preprint arXiv:2103.06784* (2021).

[8] Sébastien Bubeck. 2015. Convex optimization: Algorithms and complexity. *Foundations and Trends® in Machine Learning* 8, 3-4 (2015), 231–357.

[9] Felix Campbell, Bahareh Arab, and Boris Glavic. 2022. Efficient Answering of Historical What-if Queries. In *Proceedings of the 48th International Conference on Management of Data (SIGMOD)*. 1556–1569. https://doi.org/10.1145/3514221.3526138

[10] Marco C Campi and Simone Garatti. 2011. A sampling-and-discarding approach to chance-constrained optimization: feasibility and optimality. *Journal of optimization theory and applications* 148, 2 (2011), 257–280.

[11] Marco C Campi, Simone Garatti, and Maria Prandini. 2009. The scenario approach for systems and control design. *Annual Reviews in Control* 33, 2 (2009), 149–157.

[12] Siu On Chan, Ilias Diakonikolas, Rocco A Servedio, and Xiaorui Sun. 2014. Near-optimal density estimation in near-linear time using variable-width histograms. *Advances in neural information processing systems* 27 (2014).

[13] Michael Chau, Reynold Cheng, Ben Kao, and Jackey Ng. 2006. Uncertain data mining: An example in clustering location data. In *Advances in Knowledge Discovery and Data Mining: 10th Pacific-Asia Conference, PAKDD 2006, Singapore, April 9-12, 2006. Proceedings 10*. Springer, 199–204.

[14] Nilesh Dalvi and Dan Suciu. 2007. Efficient query evaluation on probabilistic databases. *The VLDB Journal* 16 (2007), 523–544.

[15] Michael R Garey and David S Johnson. 1979. *Computers and intractability*. Vol. 174. freeman San Francisco.

[16] James E Gentle. 2009. *Computational Statistics*. Springer.

[17] Soumyadip Ghosh and Shane G Henderson. 2003. Behavior of the NORTA method for correlated random vector generation as the dimension increases. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 13, 3 (2003), 276–294.

[18] Francesco Gullo, Giovanni Ponti, and Andrea Tagarelli. 2008. Clustering uncertain data via k-medoids. In *International Conference on Scalable Uncertainty Management*. Springer, 229–242.

[19] Francesco Gullo, Giovanni Ponti, and Andrea Tagarelli. 2013. Minimizing the variance of cluster mixture models for clustering uncertain objects. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 6, 2 (2013), 116–135.

[20] Francesco Gullo, Giovanni Ponti, Andrea Tagarelli, and Sergio Greco. 2008. A hierarchical algorithm for clustering uncertain data via an information-theoretic approach. In *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 821–826.

[21] Francesco Gullo, Giovanni Ponti, Andrea Tagarelli, and Sergio Greco. 2017. An information-theoretic approach to hierarchical clustering of uncertain data. *Information sciences* 402 (2017), 199–215.

[22] Francesco Gullo and Andrea Tagarelli. 2012. Uncertain centroid based partitional clustering of uncertain data. *arXiv preprint arXiv:1203.6401* (2012).

[23] Riddho R. Haque, Anh L. Mai, Matteo Brucato, Azza Abouzied, Peter J. Haas, and Alexandra Meliou. 2024. Stochastic SketchRefine: Scaling In-Database Decision-Making under Uncertainty to Millions of Tuples. arXiv:2411.17915 [cs.DB] https://arxiv.org/abs/2411.17915

[24] Tito Homem-de Mello and Güzin Bayraksan. 2014. Monte Carlo sampling-based methods for stochastic optimization. *Surveys in Operations Research and Management Science* 19, 1 (2014), 56–85.

[25] Shudong Huang, Zhao Kang, Zenglin Xu, and Quanhui Liu. 2021. Robust deep k-means: An effective and simple method for data clustering. *Pattern Recognition* 117 (2021), 107996.

[26] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Perez, Chris Jermaine, and Peter J. Haas. 2011. The Monte Carlo database system: Stochastic analysis close to the data. *ACM Trans. Database Syst.* 36, 3, Article 18 (2011), 41 pages. https://doi.org/10.1145/2000824.2000828

[27] Sujin Kim, Raghu Pasupathy, and Shane G Henderson. 2015. A guide to sample average approximation. In *Handbook of Simulation Optimization*. Springer, 207–243.

[28] Anh L. Mai, Pengyu Wang, Azza Abouzied, Matteo Brucato, Peter J. Haas, and Alexandra Meliou. 2024. Scaling Package Queries to a Billion Tuples via Hierarchical Partitioning and Customized Optimization. *Proc. VLDB Endow.* 17, 5 (2024), 1146–1158.

[29] Alexander J. McNeil, Rüdiger Frey, and Paul Embrechts. 2015. *Quantitative Risk Management: Concepts, Techniques and Tools*. Princeton University Press.

[30] Paul Mooney. [n.d.]. Stock Market Data (NASDAQ, NYSE, S&P500). https://www.kaggle.com/datasets/paultimothymooney/stock-market-data/data. Accessed: March 10, 2024.

[31] Victor M Panaretos and Yoav Zemel. 2019. Statistical aspects of Wasserstein distances. *Annual review of statistics and its application* 6, 1 (2019), 405–431.

[32] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71.

[33] Krishna Reddy and Vaughan Clinton. 2016. Simulating stock prices using geometric Brownian motion: Evidence from Australian companies. *Australasian Accounting, Business and Finance Journal* 10, 3 (2016), 23–47.

[34] Sergey Sarykalin, Gaia Serraino, and Stan Uryasev. 2008. Value-at-Risk vs. Conditional Value-at-Risk in Risk Management and Optimization. In *State-of-the-Art Decision-Making Tools in the Information-Intensive Age*. INFORMS TutORials in Operations Research, 270—-294. https://doi.org/10.1287/educ.1080.0052

[35] Jonathon Shlens. 2014. Notes on Kullback-Leibler divergence and likelihood. *arXiv preprint arXiv:1404.2000* (2014).

[36] 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*. 1–12.

[37] Laurynas Siksnys, Torben Bach Pedersen, Thomas Dyhre Nielsen, and Davide Frazzetto. 2021. SolveDB+: Sql-based prescriptive analytics. In *Advances in Database Technology-24th International Conference on Extending Database Technology, EDBT 2021*. OpenProceedings. org, 133–144.

[38] Eric K Tokuda, Cesar H Comin, and Luciano da F Costa. 2022. Revisiting agglomerative clustering. *Physica A: Statistical mechanics and its applications* 585 (2022), 126433.

[39] Kyoung-Gu Woo, Jeong-Hoon Lee, Myoung-Ho Kim, and Yoon-Joon Lee. 2004. FINDIT: a fast and intelligent subspace clustering algorithm using dimension voting. *Information and Software Technology* 46, 4 (2004), 255–271.

[40] Xianchao Zhang, Han Liu, and Xiaotong Zhang. 2017. Novel density-based and hierarchical density-based clustering algorithms for uncertain data. *Neural networks* 93 (2017), 240–255.