The Dangerous Impact of Solver Imprecision on Data Management Techniques (and How to Avoid It)

Zixuan Chen
zixuan.chen90@tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Zikun Wang
wang.zikun@northeastern.edu
Northeastern University
Boston, USA

Panagiotis Manolios
p.manolios@northeastern.edu
Northeastern University
Boston, USA

Mirek Riedewald
m.riedewald@northeastern.edu
Northeastern University
Boston, USA

Abstract

Linear Programming (LP, MILP, ILP) solvers have been applied to data management problems for decades, with increasing use in the context of reverse data management and algorithmic fairness. While numerical imprecision caused by floating-point arithmetic is a known tradeoff made by fast solvers, it has received almost no attention in the data management literature relying on these tools. We demonstrate analytically and empirically that this can be a costly mistake. Even when supplying a theoretically correct (MI)LP program, the solver's answer may be incorrect, with potentially severe implications. This paper proposes a general solution that relies on "gap" parameters and efficient algorithms to explore the combinatorial search space of their values. Barring problem-specific solutions, current solver technology generally forces a choice between severely limited performance (when using exact solvers) and approximate solutions with little to no non-trivial quality guarantees (when using fast solvers that rely on floating-point arithmetic).

Keywords

Linear programming, numerical imprecision, solvers

1 Introduction

Many fundamental data management problems can be modeled as Linear Programs (LP), Integer Linear Programs (ILP), or Mixed Integer Linear Programs (MILP). This includes exciting recent work, from enhancing database engines with new features [5–7, 11, 21, 24] to solving problems in rankings [2, 3, 9, 10, 12, 13], causality [22] and resource management [23]. They all focus on efficient implementations that involve state-of-the-art (SOTA) solvers[1] like Gurobi [16] and Cplex [19]. Those solvers have become fast and scalable for real-world linear programs, but at a price: *they rely on precision tolerances and imprecise floating-point arithmetic.* While this limitation is well-known and documented, including recipes for dealing with it [15], previous data management research that uses these solvers does not seem to consider it at all. (We also only recently started studying the issue [13].) We show that this oversight can result in surprising and incorrect outcomes. And we demonstrate both analytically and empirically that there is no perfect solution.

[1]These tools are also often called "optimizer", "optimization studio", etc. We use the general term "solver".

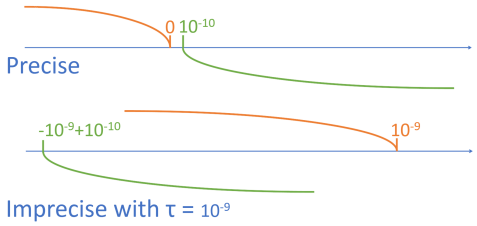


Figure 1: Numerical imprecision

Example 1 (Feasibility imprecision and feasibility tolerance). *Constraint* $x \leq 0 \wedge x \geq 10^{-10}$ *is unsatisfiable, but widely-used and SOTA solvers such as Gurobi [16], Cplex [19], Scip [1] and Scipy.optimize [26] all claim that a feasible solution* $x = 0.0$ *exists. This minimal example highlights the inherent tradeoff solvers make when relying on floating-point arithmetic for performance. As Gurobi's* Guidelines for Numerical Issues *[15] reveal, a* feasibility tolerance $\tau > 0$ *(tau) for constraints causes* $ax - b \geq 0$ *for variable* x *and constants* a, b *to be considered satisfied if* $\mathtt{a} * \mathtt{x} - \mathtt{b} \geq -\mathtt{tau}$.[a] *In our example, as illustrated in Figure 1, this leads the solver to believe that* $x \geq 10^{-10}$ *is satisfied by* $x = 0.0$ *when* $\tau = 10^{-9}$. *As we discuss below, while* τ *can be adjusted, it cannot be set to zero; and smaller values incur a performance penalty. Hence, errors like this are a fact of life when using fast solvers.*

[a]We use typewriter font and code-like syntax to distinguish a floating-point computation like $\mathtt{a} * \mathtt{x} - \mathtt{b} \geq -\mathtt{tau}$ in a solver from the intended mathematical meaning like $ax - b \geq 0$.

One may wonder if the issue in Example 1 could be addressed by compensating for the tolerance: convert each constraint to canonical form $f(\mathbf{x}) \geq 0$ and pass it to the solver as $\mathtt{f}(\mathbf{x}) \geq \mathtt{tau}$. (Here $\mathbf{x}$ denotes a vector of variables.) Although it motivates our general solution (Section 4), this idea does not make the imprecision problem go away—if it did, solvers could apply it internally. The precision tolerance is designed to counteract the imprecise nature of floating-point arithmetic, but the true error when evaluating a formula like $f(\mathbf{x})$ cannot be precisely quantified: it depends not only on the specific values of variables and constants and the number and type of operations, but also on internal floating-point computations made by the solver while deriving intermediate results. Hence, implementing $f(\mathbf{x}) \geq 0$ as $\mathtt{f}(\mathbf{x}) \geq \mathtt{tau}$ can reduce the number of false positives, i.e., variable assignments the solver incorrectly believes to satisfy a constraint, while potentially introducing false negatives, i.e., satisfying assignments the solver incorrectly rules out (Section 4). In addition to LP, which exclusively relies on continuous variables, the

EDBT '27, Lille (France)

10.48786/edbt.2027.09

problem also affects MILP (see Section 3.2) and even ILP (see Section 3.1), where all variables are integers.

This work makes the following main contributions:

- We discuss the numerical issues in fast solvers that are most relevant for data management problems (Section 2).
- We show that data management techniques relying on fast solvers are affected by their imprecision, causing incorrect results (Section 3).
- We propose a systematic approach to detect and address these problems with "gap parameters" (Section 4).
- Experiments demonstrate the effectiveness of both the gap parameters and our heuristics for setting the gap-parameter values and show that existing precise solvers do not scale to realistic data problems (Section 5).

2 Numerical Imprecision in Solvers

Solvers using floating-point arithmetic overcome the severe scalability limits of exact solvers (Section 5.4), but suffer from imprecision.

2.1 Problems Caused by Numerical Imprecision

Imprecision-related problems are caused by floating-point arithmetic, e.g., the widely adopted IEEE 754 standard. Its combination of fixed size (8 bytes for double precision) and hardware support achieves huge performance benefits at the cost of approximating the mathematical properties of the real numbers. For instance, `1 == 1 + 1e-16` and `1 + 1e16 == 1e16` evaluate to `True` across different programming languages [15]. Similarly, `(1 + 1e-16) + 1e-16 == 1 + (1e-16 + 1e-16)` evaluates to `False`, meaning floating-point arithmetic may violate associativity. Small inaccuracies can amplify during a computation, like in this program, which produces an error equal to the value of `expected`, wreaking havoc on any computation involving `f`:

```
expected = 1000.0
x = 1e-16  # below machine precision
x_0 = (1.0 + x) - 1.0  # supposed to be equal to x, but actually 0
f = expected * (x_0 / x)
error = abs(f - expected)
```

In general floating-point expression $\mathtt{f}(\mathbf{x})$ corresponds to real number $f(\mathbf{x}) + E$, where E could be 0, positive or negative. The more complex the expression is, the more the individual errors may compound. This induces *feasibility imprecision* and *optimality imprecision* in solvers that try to address the errors through the corresponding tolerance parameters.[2] As illustrated in Example 1, the feasibility tolerance slightly shifts constraints. The optimality tolerance determines "feasibility of dual constraints, i.e., whether $ay \leq c$ for constants a, c and variable y, holds for the dual solution. More precisely, $ay \leq c$ will be considered to hold if $\mathtt{a} * \mathtt{y} - \mathtt{c} \leq \mathtt{OptimalityTol}$" [15]. As illustrated in Example 2, this implies that the solver may return a solution that is slightly worse than optimal, but "close enough:"

Example 2 (Optimality imprecision). *For knapsack problem*

$$\begin{aligned} max \quad & 100000.0001x_1 + \textstyle\sum_{i=2}^{1,000,001} 0.1x_i \\ s.t. \quad & 1000000x_1 + \textstyle\sum_{i=2}^{1000001} x_i \leq 1000000 \\ & x_i \in \{0, 1\} \end{aligned}$$

the optimal answer is $x_1 = 1$ (all others zero) with objective value 100,000.0001. However, fast solvers may return $x_1 = 0$ and all others set to 1, with suboptimal objective 100,000.0.

Note that despite realizing near-identical objective values, the two variable assignments are opposites of each other. Example 2 also demonstrates that numerical problems appear even in the context of ILPs, where all variables are integer. Although the example uses non-integer coefficients in the objective, even for integer coefficients, scaling operations, e.g., applied by the user to address numerical instability [15] or by the solver during intermediate computation steps, could result in non-integer coefficients.

2.2 Optimal or Not? How Close?

Given the answer to an optimization problem returned by a fast solver, what can be inferred about the true optimum? In general, "while it is true that for most practical optimization problems, small perturbations in the input only induce small perturbations in the final answer to the problem, there are some special situations where this is not the case. These ill behaving problems are called *Ill Conditioned* or *Numerically Unstable*" [15]. Conditioning is not a binary property, but a spectrum of the quantitative impact of input perturbations on the objective. While many heuristics exist, there is no general algorithm for accurately estimating the approximation error and minimizing it—the user must custom-design a problem-specific transformation of the program.

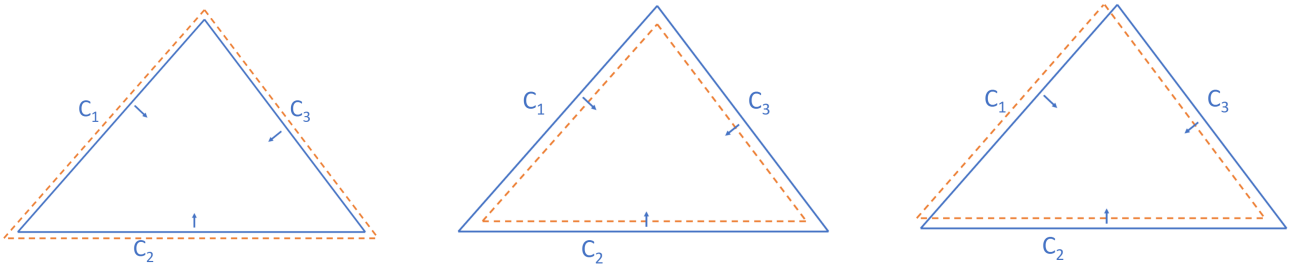
We now provide an intuition of how floating-point imprecision and the solver's precision tolerances affect the answer in a non-trivial manner. The interested reader may consult the extensive literature on the problem, starting from the solver manual [15]. Assume we are given an LP minimization problem over variables $\mathbf{x}$ where, without loss of generality, all constraints are of type $f(\mathbf{x}) \geq 0$. (Any constraint given in a different format can be converted to this canonical form.)

Impact through constraints. When $f(\mathbf{x})$ is close to 0, the solver may incorrectly consider a feasible solution infeasible (false negative) or vice versa (false positive). Worse yet, depending on internal choices made by the solver, e.g., the pre-processing or optimization algorithm it uses, the outcome may differ. The solver's feasibility tolerance τ can prevent false negatives, as long as τ exceeds the overall error. Unfortunately, a sufficiently tight upper bound for that error is challenging to obtain, because it depends on the values of variables and coefficients, the structure of f, and even the solver-internal algorithms. And for a problem with multiple constraints, a single τ may be too coarse. Hence, in general the solver's view based on floating-point arithmetic corresponds to a slightly expanded and/or reduced version of the true mathematical region of feasible solutions for $\mathbf{x}$. This means that it may miss the true optimal answer or pick a spurious one that in reality is infeasible as illustrated in Figure 2.

Impact through the objective. As shown in Example 2, when solutions have similar objective values, the worse one may overtake the better one due to imprecision. The optimality tolerance lets the solver determine when to stop searching for a better answer.

In summary, the solver may ignore the true optimum; it may pick a spurious better-than-optimum solution outside the true feasible region; and among the solutions it considers, it may not pick the best one. While this sounds pessimistic, in practice a well-conditioned problem will provide an objective close to the optimum. However, the difference may be large depending on the problem instance. And even if the objective value is close

[2] There can be other tolerances in a solver, e.g., the integer feasibility tolerance for the integrality of solutions.



(a) Relaxed constraints: false positives (b) Stricter constraints: false negatives (c) Both may happen at the same time

Figure 2: Impact of imprecise constraints on the solution space. The true space should be bounded by constraints C_1, C_2, C_3, but imprecision and tolerances may cause each constraint to be relaxed or stricter, and cause the solution space to expand, shrink or shift in all directions as indicated by the dashed lines.

to the optimum, the variables may have very different values (Example 2).

Can smaller tolerances eliminate numerical issues? Fast solvers like Gurobi and Cplex let the user control precision tolerances [17, 18]. However, because of the inherent imprecision of floating-point arithmetic, tuning of tolerances cannot eliminate the problem. In fact, even for their lowest supported feasibility tolerance $\tau = 10^{-9}$, both Gurobi and Cplex still produced the false positive from Example 1 and similar problems we explored.

3 Impact on Database Problems

To demonstrate that issues caused by numerical imprecision are not just hypothetical, we study 2 recent solver uses for data management applications.

3.1 Package Queries

Package queries [5–8, 21] utilize fast solvers for knapsack-style ILP programs to identify a set of input tuples that optimizes a linear objective while enforcing linear constraints over the tuples in the package. For example, users can express "find a set of 3 gluten-free meals with 2,000 to 2,500 combined calories that has the minimal total amount of fat" with an SQL-inspired query language. The execution engine converts it to an ILP program, relying on a fast solver to compute the optimal answer. In our empirical evaluation we started with the original queries and data from recent work [21], slightly modifying query constants or individual tuples to determine if a minor change could trigger incorrect results. The query below searches for groups of stars in a partition of the galaxy-view data in the Sloan Digital Sky Survey (SDSS) [25]. It minimizes the total mass T_mass, subject to constraints on group cardinality and sums over the attributes J, H, K:

$$\begin{array}{lll} \min & \sum_{i=1}^{n} x_i \cdot T\_mass_i & \\ \text{s.t.} & \sum_{i=1}^{n} x_i \geq 15 & \sum_{i=1}^{n} x_i \leq 45 \\ & \sum_{i=1}^{n} J_i \cdot x_i \geq b_1 & \sum_{i=1}^{n} H_i \cdot x_i \leq b_2 \\ & \sum_{i=1}^{n} K_i \cdot x_i \geq b_3 & \sum_{i=1}^{n} K_i \cdot x_i \leq b_4 \end{array} \quad (1)$$

The parameters b_1, b_2, b_3, b_4 enable the creation of versions of the query. Similar to Example 2, our empirical evaluation shows that despite all variables x_i being integer, feasibility and optimality imprecision can lead to spurious or suboptimal answers (Section 5.1). Even for tiny differences in the objective value, the technique might deliver a completely different package!

3.2 Ranking Problems

Recent work on explanations, fairness and diversity for top-k queries and rankings employs fast solvers to synthesize linear scoring functions that satisfy constraints enforcing goals such as representation, robustness, or approximation of a given ranking [2, 3, 11–13]. The discrete nature of a ranking can amplify tiny imprecisions. Let δ_{sr} be the hyperplane of weight vectors—each vector defining a linear scoring function—for which tuples s and r have identical scores. For vectors above δ_{sr}, s has the higher score and thus would be ranked above r; and vice versa for vectors below δ_{sr}. Imprecision can "teleport" a weight vector close to δ_{sr} to the wrong side of the hyperplane. This means that while the solver believes s to have a higher score than r, in reality the order is reversed, turning a minuscule score difference into a full order inversion. Common data properties can aggravate the situation. For example, tuples with similar attribute values induce similar hyperplanes, thus a weight vector might teleport through multiple hyperplanes at once. Imprecision can also eliminate thin feasible regions (and thus potentially the optimal solution) or create spurious regions similar to Example 1. These problems affect not only approaches that explicitly rely on the hyperplanes to partition the search space [2, 3], but also those that use indicators [12, 13]. In our empirical evaluation (Section 5.2), we observed a massive impact from spurious regions.

4 Dealing with Numerical Issues

We now propose the first general solution for data management techniques that rely on fast solvers by addressing both feasibility imprecision and optimality imprecision. The former refers to the discrepancy between a variable assignment truly satisfying a constraint vs the solver believing that it does (Example 1). The latter means that the solver could return a variable assignment that, even though it achieves a near-optimal objective value, differs from the true optimal assignment (Example 2). It is important to keep the following in mind:

Lemma 1 (Unknown Optimal Answer). *When using fast imprecise solvers, we generally do not know the true optimal objective value nor the variable assignment(s) that achieves it.*

4.1 Addressing Optimality Imprecision

Lemma 1 implies that given a variable assignment $\mathbf{X}$ with objective value o returned by a fast solver, one cannot determine how close o is to the true optimum or how similar $\mathbf{X}$ is to the optimal assignment. This eliminates any type of directional or gradient-based search procedure, leaving only grid search to explore diverse alternative assignments: Partition the space defined

Algorithm 1: Gap-parameter grid search with verification

```
Input: (MI)LP P(ε); grid points cand(ε_i)_{i=1,...,m}
Output: best verified solution of P found
best_solution = NULL;
foreach grid point g ∈ cand(ε_1) × ··· × cand(ε_m) do
    Solution X = run fast solver on P(ε = g);
    if X passes verifier then
        best_solution = winner( best_solution, X )
Return best_solution
```

by the variables into a grid, run the solver for each grid cell, pick the winner among them and compare it with the optimal objective value when the solver is executed over the entire space.

4.2 Addressing Feasibility Imprecision

4.2.1 Detection through Exact Verification. While it is not possible to determine if feasibility imprecision resulted in a suboptimal answer (Lemma 1), we can check if the variable assignment $\mathbf{X}$ returned by a solver satisfies all constraints. To this end, one can select from a variety of exact-arithmetic libraries like *BigDecimal* and *BigFraction* in Java or *decimal* and *fractions* in Python. Even when the cost of exactly solving an optimization problem is prohibitive, checking linear constraints is very fast. To verify $\mathbf{X}$, first retrieve the values of all variables from the solver. Then evaluate each constraint one-by-one, reporting those that fail verification. We can similarly evaluate the objective function to determine its exact value. However, a match or mismatch between the exact objective value and the value returned by the solver does not prove or disprove the optimality of $\mathbf{X}$. Based on this verification process, which (1) determines if the given assignment $\mathbf{X}$ is valid, i.e., satisfies all constraints, and (2) returns the exact value of the objective function for $\mathbf{X}$ (instead of the solver-reported one), we design an algorithm that automatically searches for better solutions by exploring gap parameters.

4.2.2 Gap Parameters. For the remainder of the discussion, let P denote the original linear program with all its m constraints in canonical form, i.e., $f_j(\mathbf{x}) \geq 0, j = 1, \ldots, m$. If any $f_i(\mathbf{x}) \geq 0$ fails verification, i.e., the solver believes $\texttt{f\_i}(\mathbf{x}) \geq 0$ but the verifier determines that $f_i(\mathbf{x}) < 0$, we tighten the constraint to $\texttt{f\_i}(\mathbf{x}) \geq \varepsilon\_\texttt{i}$, using a *gap parameter* $\varepsilon_i \geq 0$. Define $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_m)$ and let $P(\boldsymbol{\varepsilon})$ be program P with each constraint $f_j(\mathbf{x}) \geq 0$ replaced by the tighter version $\texttt{f\_j}(\mathbf{x}) \geq \varepsilon\_\texttt{j}$. For a sufficiently large ε_j that overcomes the solver imprecision, the assignment will pass verification of the original constraint $f_j(\mathbf{x}) \geq 0$. However, setting ε_j too high may eliminate the true optimum.

Unfortunately, there is no well-defined "optimal" value for each individual ε_j. In general, it could depend on the assignment $\mathbf{X}$, which in turn is affected by the gap parameters of all other constraints. Hence, one must holistically "guess" the entire vector $\boldsymbol{\varepsilon}$. This, like for optimality imprecision, implies the need for grid search, but on the values of $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_m)$, instead of the input variables. Algorithm 1 shows the implementation of this idea. Given candidate values for each gap parameter, it solves the corresponding optimization problem and keeps track of the best assignment that passed the verification.

4.2.3 Practical Gap-Parameter Tuning. Like for grid search in other contexts, such as hyper-parameter tuning for machine-learning models, it is doubtful that a general algorithm with optimality or approximation guarantees can be found for gap-parameter tuning. Hence, we propose heuristics and provide practical suggestions. For better exposition, assume P has 4 constraints and the user wants to explore 10,000 gap-parameter combinations.

Adaptive search exploits an approximate notion of monotonicity of gap parameters: larger values for ε_i tend to reduce the number of false positives (assignments returned by the solver that fail verification), while potentially introducing more false negatives (valid assignments excluded). This holds only in the sense of a correlation, i.e., one generally cannot exclude the possibility that the assignment obtained for a smaller "lucky" ε_i passes verification, while the assignment obtained for a larger ε_i fails (even when holding all other gap parameters constant). The heuristic starts with a coarser grid and then refines it in promising regions. The idea is that around a successfully verified solution, there may be others that also pass verification, but potentially improve the objective, especially when their gap-parameter values are slightly lower. In the example, 10,000 combinations could be generated as follows: Start with a coarse grid of 7 values per gap parameter, for a total of 2,401 combinations. For the top-3 (in terms of the objective value) that pass verification, apply the same 7-values-per-dimension idea, but now to the neighborhood around each of these top-3 candidates.

Geometric sequence. It is important to identify the right order of magnitude for each gap parameter. Hence we recommend a geometric sequence of values in each dimension. For example, to explore 4 values between 0.001 and 1.0, use (0.001, 0.01, 0.1, 1.0) instead of (0.001, 0.34, 0.67, 1.0). This dovetails with adaptive search.

Dimensionality reduction. Inspired by the single global feasibility tolerance of Gurobi, we can use a single gap parameter ε, i.e., set $\varepsilon = \varepsilon_1 = \varepsilon_2 = \cdots = \varepsilon_m$. This trades flexibility for more fine-grained coverage of gap values. In the example, instead of trying 10 different values in each of the 4 dimensions, one can now check 10,000 values for the single gap parameter. Between the extremes, one can exploit problem symmetries to identify groups of related constraints and assign the same gap parameter to each group member as the following case study illustrates.

4.2.4 Case Study: Ranking with Ties. Consider the problem of ranking tuples by their scores, where $\text{score}(r)$ is a floating point number computed as a weighted sum over r's attribute values. Since it is not meaningful to check for equality of floating point numbers, tuples r, s are considered tied iff $-\alpha \leq \text{score}(r) - \text{score}(s) \leq \alpha$, for some small precision tolerance $\alpha > 0$. SOTA linear programs for ranking problems rely on indicator variables [11–13] to model tuple ordering. For tuple pair (r, s), $\delta_{rs} = 1$ indicates that r beats s and $\delta_{rs} = 0$ indicates that it does not. Hence $(\delta_{rs} = 1, \delta_{sr} = 0)$ when r beats s, $(\delta_{rs} = 0, \delta_{sr} = 1)$ when s beats r, $(\delta_{rs} = 0, \delta_{sr} = 0)$ when r and s are tied, and $(\delta_{rs} = 1, \delta_{sr} = 1)$ is invalid.

This results in the following constraints in the linear program for each tuple pair (r, s) of interest:

$$\begin{aligned} \delta_{rs} = 1 \quad &\Rightarrow \quad \text{score}(r) - \text{score}(s) > \alpha \\ \delta_{rs} = 0 \quad &\Rightarrow \quad \text{score}(r) - \text{score}(s) \leq \alpha \end{aligned} \tag{2}$$

Converting to canonical form $f(\mathbf{x}) \geq 0$ and replacing 0 by the gap parameter, we obtain

$$\begin{aligned} \delta_{rs} = 1 \quad &\Rightarrow \quad \text{score}(r) - \text{score}(s) - \alpha^+ \geq \varepsilon_{1,rs} \\ \delta_{rs} = 0 \quad &\Rightarrow \quad -\text{score}(r) + \text{score}(s) + \alpha \geq \varepsilon_{2,rs} \end{aligned} \tag{3}$$

Here α^+ denotes a value marginally greater than α to convert the strict inequality to a non-strict one.

For 100 tuples and the problem of approximating a top-5 ranking, there are 990 constraints. Even trying only 2 candidate values for each gap parameter would result in a grid with 2^{990} combinations, requiring dimensionality reduction. Exploiting symmetry, a natural approach could be to assign the same ε_1 to all constraints for indicator value 1, and ε_2 to all constraints with indicator value 0. This may be preferable over using the same ε for all constraints, because a larger gap may eliminate ties: From the constraints for $(\delta_{rs} = 0, \delta_{sr} = 0)$ we derive $2\alpha \geq \varepsilon_{2,rs} + \varepsilon_{2,sr}$, establishing an upper bound on the corresponding gap values.

5 Experiments

We present evidence that solver imprecision impacts data management applications (Sections 5.1 and 5.2), that our gap parameters are effective (Section 5.3), and that exact solvers are not viable for realistic data management applications (Section 5.4).

Setup. The ranking experiments are built on the open-source code provided by [13] and tested in Java 18. The package query and scalability experiments are tested in Python 3.13.11. We use library/package versions 13.0.0 for Gurobi and 4.16.0.0 for Z3. All experiments are executed on a Windows 11 laptop with an Intel(R) Core(TM) Ultra 9 275HV processor and 32GB RAM.

5.1 Impact on Package Queries

We solve Equation (1) for different values of b_1, b_2, b_3, b_4. Let X_1 be the assignment returned by the solver. If it fails verification, we have evidence of *feasibility imprecision*. If it passes, we adjust b_1 so that X_1 lies by 10^{-6} *outside* the feasible region, i.e., it should now be infeasible. If the solver still returns it as the optimal answer, this also provides evidence for feasibility imprecision. In [21], 4 values are provided for each of b_1, b_2, b_3, b_4, resulting in 256 combinations. We create 100 test datasets, each by randomly sampling 45 tuples from the original dataset. Then we apply each of the 256 constraint variants to all 100 test sets to check for feasibility imprecision whenever a solution exists. Out of the 25,600 runs, X_1 fails verification in 8 of them, affecting 2 of the 100 test sets. When adjusting b_1 in the others to exclude X_1, in 444 cases the solver still returns the now invalid assignment, affecting 73 of the 100 test sets.

To demonstrate *optimality imprecision* without knowing the true optimum (Lemma 1), we designed the following approach: After receiving X_1, remove one of its selected tuples from the input, forcing a different solution X_2 when running the solver again. Using exact arithmetic, check if X_1 and X_2 have the same objective value. If they do, restore the original input and make X_1 slightly worse than X_2 by adding 10^{-7} to T_mass of a tuple that appears in X_1, but not X_2. If the solver still returns X_1 on the modified input, it provides evidence of optimality imprecision. For example, on the original input X_1 and X_2 both had objective value 62.5, but the solver believed it to be 62.4999997 for X_1 and 62.4999999 for X_2. After the input modification, the solver's belief of X_1's objective increased to 62.4999998, still seemingly beating X_2. Using the methodology from the feasibility-imprecision experiment, we created 1000 test sets. Out of the 256,000 runs, we found suitable pairs X_1, X_2 in 356 of them. After injecting 10^{-7} to make X_1 worse than X_2, the solver still returned X_1 in 16 runs, affecting 2 test sets.

While the number of errors caused by imprecision may appear low, note that (1) they include only cases where we could *prove* their impact and (2) even a single occurrence demands a solution.

5.2 Impact on Ranking Problems

We first quantify the appearance of *feasibility imprecision* in indicator-based programs that synthesize a scoring function for a given top-5 ranking [13] on real datasets NBA [20] and CSRankings [4]. From each dataset, 100 test sets are generated, each by sampling 5 attributes and 100 tuples. When ignoring numerical issues, the numbers are staggering: In 16% of NBA and 72% of CSRankings instances, the solver's view differed from the true ranking!

We then study the impact of *feasibility imprecision* on partition-based approaches to ranking problems [3]. Partitions are defined by recursively splitting the problem space with hyperplanes equivalent to the indicators: linear scoring functions where tuple r beats s lie above the hyperplane, those where s beats r lie below, and ties lie right on it. Due to the high computational cost, we use a small real dataset with the player stats (8 attributes) of the top-8 NBA players in the 2023/24 MVP ranking. There are 28 unique hyperplanes, one per player pair. After each split, the solver is used to check if the corresponding intersection of half-spaces exists, otherwise terminating the branch. The approach constructs a tree of 99,593 nodes and solves 162,521 linear programs. Our verification shows that among the 49,797 leaf nodes, the solver selects a bad representative (a point outside the leaf's region) for 49,585 of them—*99.57%*! This is partly caused by the solver picking solutions near the boundary, but possibly also by spurious regions due to *feasibility imprecision*. Applying our gap parameters, execution time drops to 1/10 as the resulting tree only has 6,623 nodes and solves 15,890 linear programs. Now all 3,312 leaf nodes are valid regions and the solver picks points truly inside them.

5.3 Effectiveness of Gap Parameters

We applied Algorithm 1 to the indicator-based programs for approximating a top-5 ranking (Section 5.2). When synthesizing a scoring function, an assignment that fails verification is still a valid scoring function. It simply does not produce the ranking the solver thought it would. When picking an assignment that appears better, but in reality leads to a worse ranking, imprecision introduces an error. After running the 100 programs for each dataset, we report (1) how many of them failed verification and (2) the relative error compared to the best ranking found. We set tie tolerance $\alpha = 10^{-3}$.

Table 1 summarizes the main observations. When ignoring numerical issues (original problem), 16 and 72 out of 100 problem instances fail verification for NBA and CSRankings, respectively. This leads to a greater ranking-approximation error. In contrast, all variants of applying our gap parameters completely eliminated verification failures and extra approximation error compared to the best scoring functions found across all methods. Here it did not matter if we used a single gap parameter for all constraints (geometric sequence of 90 grid points between 10^{-10} and 10^{-1}) or 2 gap parameters depending on the indicator's value (10-by-10 grid with values between 10^{-10} and 10^{-1}); or if we used adaptive grid search (start with 10 points in range 10^{-10} and 10^{-1}, then refine twice with 10 new grid points around the best initial value). This demonstrates the robustness of Algorithm 1.

Table 1: Impact of gap parameters

Approach	Dataset	Incorrect objective	Extra error	Number of grid points	Avg runtime
Original	NBA	16%	13.1%	1	126ms
Single gap parameter	NBA	0%	0	90	11217ms
Two gap parameters	NBA	0%	0	100	13065ms
Adaptive search with single gap parameter	NBA	0%	0	31	3967ms
Original	CSRankings	72%	21.7%	1	32ms
Single gap parameter	CSRankings	0%	0	90	2969ms
Two gap parameters	CSRankings	0%	0	100	3311ms
Adaptive search with single gap parameter	CSRankings	0%	0	31	1023ms

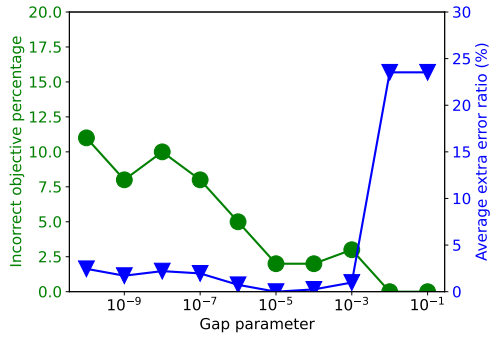


Figure 3: Impact of the gap parameter

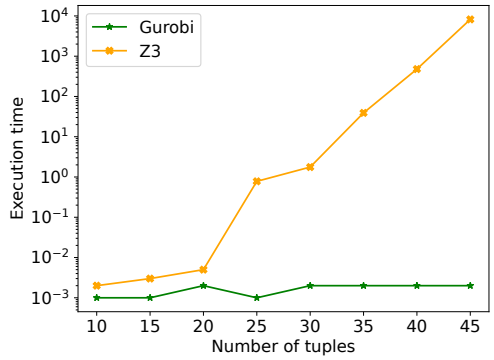


Figure 4: Precise solver vs imprecise solver

Figure 3 shows more details for the single-gap-parameter experiments. As expected, increasing the gap value reduces the number of failed verifications. This initially lowers the ranking approximation error because the solver does not pick bad scoring functions that *appear* better than they are. However, for gap values beyond 10^{-3} the error increases as good solutions are eliminated (false negatives). Here the large gap value eliminates ties, thus penalizing rankings containing tied tuples.

5.4 Why Not Use Exact Solvers?

Exact solvers like Z3 [14] can find a true optimal answer for certain problems but do not scale to realistic data management problems. In this representative experiment, we compare the execution time of a fast imprecise solver (Gurobi) to an exact solver (Z3) on synthetically generated knapsack problems. Each tuple in the dataset has an integer weight and value, drawn uniformly at random from range $[1, 10]$ and $[1, 100]$, respectively. The problem asks for a set of tuples with maximum total value that does not exceed the weight capacity (set to 100). We vary the number of input tuples and report the execution time, which is an average over 3 executions for each solver. Figure 4, with a log-scaled y-axis, shows that the exact solver (Z3) is much slower and scales poorly as input size increases. It needed more than 2 hours for a problem with 45 input tuples, while Gurobi solved it on a dataset of 100,000 tuples in 1 sec.

6 Conclusion

This paper represents an initial step toward a deeper understanding of how the imprecision of fast solvers affects the data management techniques that rely on them. We propose the first systematic solution that uses gap parameters for automatically navigating the tradeoff between not eliminating the true optimum and introducing spurious solutions that violate some constraints. Our empirical studies demonstrate that the issue "is real" and that our gap-parameter-based approach addresses it. Given the rising popularity of (MI)LP solvers for exciting data management research from reverse data management to ranking fairness, our results contain an important lesson: It is not sufficient to propose a theoretically correct (MI)LP solution, unless one wants to wait for exact solvers to dramatically improve their performance. Data management techniques relying on fast imprecise solvers must study the impact of imprecision, must verify results, and must determine if the solution found is acceptable.

Acknowledgments

This work is based upon work supported by the National Science Foundation under Grant No. 1956096 and was supported by the LOEWE Spitzenprofessur of the state of Hesse (III 5-519/05.00.003-(0005)) and the NSF grants 2517201 and 2513656.

Artifacts

The ranking experiments are built on the open-source code provided by [13] at https://github.com/northeastern-datalab/rankhow. We provide the code for the package query and scalability experiments online at https://github.com/northeastern-datalab/numerical.

References

[1] Tobias Achterberg. 2009. SCIP: solving constraint integer programs. *Mathematical Programming Computation* 1, 1 (2009), 1–41. doi:10.1007/s12532-008-0001-1

[2] Abolfazl Asudeh, H. V. Jagadish, Gerome Miklau, and Julia Stoyanovich. 2018. On Obtaining Stable Rankings. *Proc. VLDB Endow.* 12, 3 (2018), 237–250. doi:10.14778/3291264.3291269

[3] Abolfazl Asudeh, H. V. Jagadish, Julia Stoyanovich, and Gautam Das. 2019. Designing Fair Ranking Schemes. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD 2019*. 1259–1276. doi:10.1145/3299869.3300079

[4] Emery Berger. 2024. CSRankings. https://github.com/emeryberger/CSrankings, visited on July 15, 2024.

[5] Matteo Brucato, Azza Abouzied, and Alexandra Meliou. 2018. Package queries: efficient and scalable computation of high-order constraints. *VLDB J.* 27, 5 (2018), 693–718. doi:10.1007/S00778-017-0483-4

[6] Matteo Brucato, Juan Felipe Beltran, Azza Abouzied, and Alexandra Meliou. 2016. Scalable Package Queries in Relational Database Systems. *Proc. VLDB Endow.* 9, 7 (2016), 576–587. doi:10.14778/2904483.2904489

[7] Matteo Brucato, Rahul Ramakrishna, Azza Abouzied, and Alexandra Meliou. 2014. PackageBuilder: From Tuples to Packages. *Proc. VLDB Endow.* 7, 13 (2014), 1593–1596. doi:10.14778/2733004.2733038

[8] Matteo Brucato, Nishant Yadav, Azza Abouzied, Peter J. Haas, and Alexandra Meliou. 2020. Stochastic Package Queries in Probabilistic Databases. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD 2020*. 269–283. doi:10.1145/3318464.3389765

[9] Felix S. Campbell, Alon Silberstein, Julia Stoyanovich, and Yuval Moskovitch. 2024. Query Refinement for Diverse Top-k Selection. *Proc. ACM Manag. Data* 2, 3 (2024), 166. doi:10.1145/3654969

[10] Felix S. Campbell, Julia Stoyanovich, and Yuval Moskovitch. 2024. Rodeo: Making Refinements for Diverse Top-k Queries. *Proc. VLDB Endow.* 17, 12 (2024), 4341–4344. doi:10.14778/3685800.3685870

[11] Zixuan Chen, Jinyang Li, H. V. Jagadish, and Mirek Riedewald. 2025. GooseDB: A Database Engine that Optimally Refines Top-k Queries to Satisfy Representation Constraints. *Proc. VLDB Endow.* 18, 12 (2025), 5351–5354. doi:10.14778/3750601.3750669

[12] Zixuan Chen, Panagiotis Manolios, and Mirek Riedewald. 2023. Why Not Yet: Fixing a Top-k Ranking that Is Not Fair to Individuals. *Proc. VLDB Endow.* 16, 9 (2023), 2377–2390. doi:10.14778/3598581.3598606

[13] Zixuan Chen, Panagiotis Manolios, and Mirek Riedewald. 2025. Synthesizing Scoring Functions for Rankings using Symbolic Gradient Descent. In *2025 IEEE 41st International Conference on Data Engineering, ICDE 2025*. 3849–3862. doi:10.1109/ICDE65448.2025.00287

[14] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*. 337–340. doi:10.1007/978-3-540-78800-3_24

[15] Gurobi Optimization, LLC. 2025. Guidelines for Numerical Issues. https://docs.gurobi.com/projects/optimizer/en/current/concepts/numericguide.html

[16] Gurobi Optimization, LLC. 2025. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[17] Gurobi Optimization, LLC. 2025. Tolerances and user-scaling. https://docs.gurobi.com/projects/optimizer/en/current/concepts/numericguide/tolerances_scaling.html

[18] IBM. 2025. Feasibility tolerance. https://www.ibm.com/docs/en/icos/22.1.1?topic=parameters-feasibility-tolerance

[19] IBM. 2025. IBM ILOG CPLEX Optimization Studio. https://www.ibm.com/products/ilog-cplex-optimization-studio

[20] Sports Reference LLC. 2024. Basketball-Reference.com - Basketball Statistics and History. https://www.basketball-reference.com/, visited on February 17, 2024.

[21] 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. doi:10.14778/3641204.3641222

[22] Neha Makhija and Wolfgang Gatterbauer. 2023. A Unified Approach for Resilience and Causal Responsibility with Integer Linear Programming (ILP) and LP Relaxations. *Proc. ACM Manag. Data* 1, 4 (2023), 228:1–228:27. doi:10.1145/3626715

[23] Kabir Nagrecha and Arun Kumar. 2023. Saturn: An Optimized Data System for Multi-Large-Model Deep Learning Workloads. *Proc. VLDB Endow.* 17, 4 (2023), 712–725. doi:10.14778/3636218.3636227

[24] Laurynas Siksnys 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, SSDBM 2016*. 14:1–14:12. doi:10.1145/2949689.2949693

[25] Sloan Digital Sky Survey. 2025. The Sloan Digital Sky Survey. http://www.sdss.org/. Accessed: 2025-07-14.

[26] Pauli Virtanen et al. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. doi:10.1038/s41592-019-0686-2