

GooseDB: A Database Engine that Optimally Refines Top- k Queries to Satisfy Representation Constraints

Zixuan Chen
Northeastern University
Boston, USA
chen.zixu@northeastern.edu

H. V. Jagadish
University of Michigan
Ann Arbor, USA
jag@umich.edu

Jinyang Li
University of Michigan
Ann Arbor, USA
jinyli@umich.edu

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

ABSTRACT

In many applications, from university rankings to the selection of candidates for a job interview, there exist various “reasonable” ways to filter the data and generate a ranking. When the initial choice lacks certain desirable properties, we want to identify a minimally modified alternative that has those properties. To this end, we demonstrate GOOSEDB, a database engine that combines DuckDB with an MILP solver. Given an SQL query, constraints on the output, and modification preferences, GOOSEDB returns a minimally modified SQL query that satisfies the constraints. This demo focuses on representation constraints for top- k queries, i.e., count constraints over groups of tuples, such as the gender distribution of the top- k job candidates. GOOSEDB significantly generalizes previous work in two directions. First, it supports more general modifications of the selection condition and the scoring function. Second, it is the first solution to holistically optimize for both at the same time, as well as for alternative values of limit k . Conference attendees will be able to interactively refine queries from easy-to-understand applications, observing the impact of their choices.

PVLDB Reference Format:

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

1 INTRODUCTION

Rankings play an essential role for decision making, e.g., to select applicants for an interview. In practice, even the most well-intentioned ranking technique may cause undesirable outcomes, such as selecting too few qualified candidates from a certain population group. We present GOOSEDB, which formalizes *ranking* as a top- k SQL query and *representation constraints* as upper and/or lower bounds on the cardinality of groups of tuples in the k highest ranks. In response to such constraints, previous work on optimal

query refinement would adjust either the (constants appearing in) selection condition [2, 6, 9] or the scoring function [3], but not both. The following example highlights the need for a comprehensive query-refinement system that holistically considers all relevant clauses of the SQL query.

Example 1.1. Table 1 shows a dataset of 10 applicants for a tech job, with attributes ID, gender, major, GPA, and internship months. Since Gender is a protected attribute that denotes membership in a demographic group and ID is artificial, only the three remaining attributes can be used to pick interview candidates. For instance, one may filter applicants based on their Major, while relying on numeric attributes GPA and internship months for ranking. Assume one starts with the following query that selects the two CS majors with the highest GPA:

```
Q1: SELECT ID FROM APPLICANT
     WHERE Major = 'CS'
     ORDER BY GPA DESC LIMIT 2
```

As indicated in Table 1, Q1 selects applicants 6 and 7. Notably, no female applicant is chosen despite half of the applicants being female. While a CS degree and high GPA clearly matter for the job, the company is now wondering if a slightly modified query could still capture the required expertise, while also including a female applicant. This goal can be achieved in many ways (corresponding to Q2, Q3, Q4 in Table 1): (1) change the selection predicate to include EE majors, (2) modify the ranking function to reward internship experience, or (3) increase the limit to 3 candidates. No previous work supports all these types of refinement together.

The need for unified holistic query refinement becomes even more apparent when considering stricter diversity requirements, such as selecting at least 2 female candidates. In such cases, a combination of adjustments to selection conditions, ranking functions, and limits may be necessary, e.g.,

```
Q5: SELECT ID FROM APPLICANT
     WHERE Major = 'CS' OR Major = 'EE'
     ORDER BY 0.97 * GPA
            + 0.03 * Internship_months DESC LIMIT 3
```

GOOSEDB can find any of the above query refinements—selection predicates, ranking functions, and limits—holistically, based on the user’s preferences. It is built on top of DuckDB [8], introducing a new REFINE statement that allows the user to add representation

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750669

ID	Gender	Major	GPA	Internship Months	Q1	Q2: Major IN {CS, EE}	Q3: score=0.9 GPA + 0.1 Months	Q4: k = 3	Q5
1	F	CS	3.7	8			✓	✓	✓
2	F	EE	4.0	0		✓			✓
3	F	EE	3.7	2					
4	F	ME	3.5	0					
5	F	ME	3.5	3					
6	M	CS	3.9	8	✓	✓	✓	✓	✓
7	M	CS	3.8	4	✓			✓	
8	M	CS	3.6	6					
9	M	EE	3.8	0					
10	M	ME	3.6	12					

Table 1: Applicants table with primary key ID. Applicants selected by a query are marked with ✓.

constraints or general cardinality constraints over groups of tuples and to specify the attributes considered for selection and ranking. GOOSEDB interacts with the user by returning the refined query and its output, which now satisfies the constraints. Moreover, the user can specify penalties for the different types of refinement, thus guiding GOOSEDB to prefer the corresponding modifications.

To summarize, we will present GOOSEDB, a data management system that optimally refines a given SQL query based on user-supplied cardinality constraints. Compared to previous work, it supports significantly more general modifications from the combined space of selection predicates, scoring functions, and limit settings. For scoring-function refinement, it also generalizes [3] to support representation constraints on groups. For selection-predicate refinement, it extends [2] to support query-level refinements and an indicator for relaxing or tightening predicate conditions.

2 TECHNICAL BACKGROUND

2.1 Problem

Query. GOOSEDB can support Select-Project-Join-Union (SPJU) queries with an optional ORDER BY clause to sort results by a linear scoring function over numerical attributes. Similar to the state of the art [2], join and union operators are applied first, then GOOSEDB refines the resulting Select-Project (SP) query. For selection conditions over categorical attributes, any subset of values can be specified. Numerical attributes support range constraints through operators $\{\geq, =, \leq\}$.

Group cardinality constraint. A cardinality constraint over a group G of tuples enforces upper and/or lower bounds on the number of group members in the top- k . The group is defined by a selection predicate or a conjunction of multiple selection predicates. Such a constraint is commonly used to enhance the diversity or fairness of rankings, or query results in general. For example, the cardinality constraint in Example 1.1 is $|G_{Gender=F}| \geq 2$.

Refinement. We propose SQL-query refinement that includes any combination of the selection predicates (WHERE), scoring function (ORDER BY) and output size (LIMIT) in a query:

- (1) We support refinement of numerical and categorical attribute types through the modification of constants appearing in the selection condition [2, 7]. In addition, GOOSEDB supports *query-level* refinements, which can also add or remove predicates from the selection condition, thus significantly expanding refinement options.

- (2) For scoring functions, the refinements are changes to the coefficients used in the function.
- (3) For k , the refinement is the change to the output size.

The overall refinement penalty is a weighted sum over all three types of refinements, with weights determined by the user.

Optimization objective. Given a relation R , a query Q , and a constraint set C , our goal is to find a minimally modified refined query Q' whose output satisfies C . We define the modification penalty as the distance between Q and Q' as

$$\Delta(Q, Q') = a\Delta(P_Q, P_{Q'}) + b\Delta(W_Q, W_{Q'}) + c\Delta(k_Q, k_{Q'}). \quad (1)$$

Here $\Delta(Q, Q')$ is a weighted sum of the distances between selection predicates (P), scoring-function weights (W), and limit (k). The weights a , b , and c are chosen based on user preferences, with all defaulting to equal values in GOOSEDB. For example, if the user only wants to increase the limit, a and b can be set to large values to prevent refinements to the selection predicates or the scoring function.

2.2 Related Work

Early research on query refinement [4, 7] primarily explored controlling the overall size of the output. With increasing interest in diversity and fairness, recent work refines queries to satisfy cardinality constraints over groups [2, 6, 9]. However, these methods concentrate on refining selection predicates, leaving other aspects of the query largely unchanged. GOOSEDB supports more general query-level selection-predicate modifications as well as modifications to the scoring function and the output size.

Query refinement is related to explaining missing tuples in top- k queries [3, 10] and fair ranking [1]. [1, 3] identify linear scoring functions to include specific tuples in the top- k , without modifying the selection predicates. [10] considers more general refinements, but relies on sampling and greedy search, thus not producing optimal solutions.

3 SYSTEM OVERVIEW

GOOSEDB is implemented in Java 18. For SQL execution it relies on the DuckDB [8] database engine. Our new REFINE statement is transformed into an efficient Mixed-Integer Linear Program (MILP), which GOOSEDB solves using Gurobi [5], a state-of-the-art commercial solver. Similar to DuckDB, GOOSEDB operates via a command-line interface, providing users with straightforward and easy access.

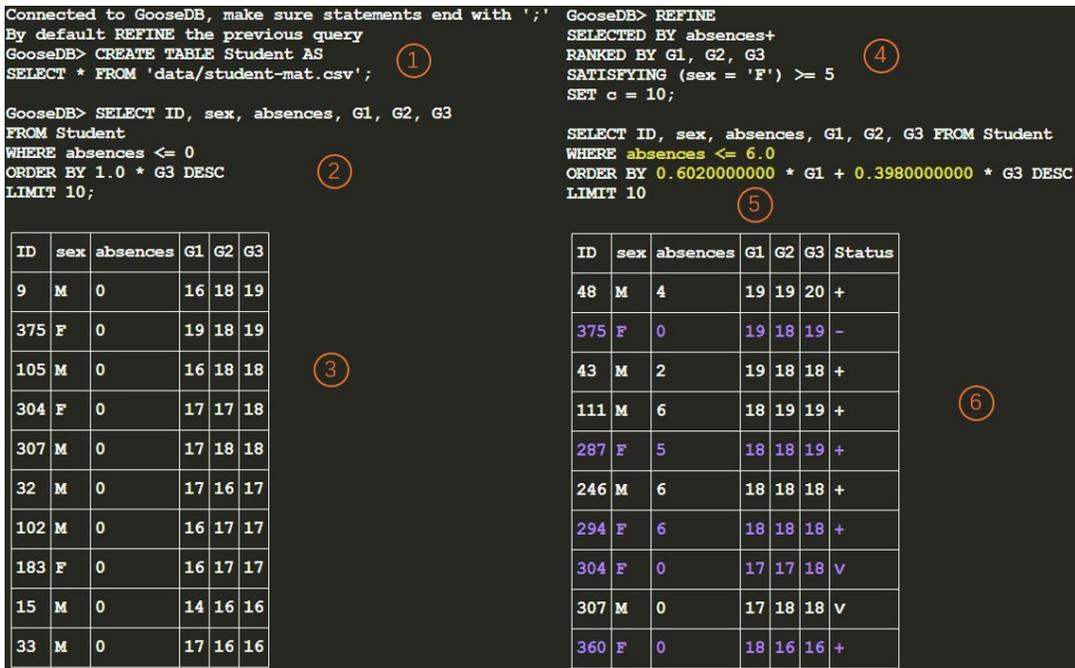


Figure 1: Interface of GooseDB

3.1 Interfaces

We present GOOSEDB’s command-line interface and a user-friendly query builder interface GOOSEFACE. Figures 1 and 2 shows how to interact with them respectively and present two user scenarios we can demonstrate. Like with a regular database, users can create tables (1, A), execute SQL statements (2, B) and obtain query results (3, C).

REFINE statement. The new REFINE statement modifies the query so that it satisfies the specified constraints. The REFINE statement consists of four key components: SELECTED BY states which attributes can be used in selection predicates, with an optional symbol to indicate relaxing (+) or tightening (−) the condition; RANKED BY specifies the attributes applicable in the scoring function; SATISFYING defines the constraints for the query result; and SET is an optional component to configure optimization parameters. Users can directly write the REFINE statement (4, D) or use the REFINE Builder in GOOSEFACE (G) to generate a REFINE statement: E corresponds to SATISFYING, the combo boxes in F correspond to SELECTED BY and RANKED BY, the slide bars (range [0, 1] in GOOSEFACE for better visualization) in F correspond to SET. Clicking G generates a REFINE statement in D.

Refinement. Given a REFINE statement, GOOSEDB computes the optimal refinement, displays the refined query (5, H) and executes it to display the output (6, I) with tuples in the specified groups highlighted (J). To help the user see the ranking changes, in GOOSEDB an extra column indicates tuple status: + for a new top- k tuple, − for no change, \wedge for a higher rank, \vee for a lower rank. In GOOSEFACE, selecting any tuple in one of the rankings highlights that tuple in both rankings.

3.2 Implementation

We introduce solution highlights, omitting technical details such as the MILP program formalization. For modification of selection predicates, we use indicator variables that model if a tuple satisfies a predicate, similar to [2]. However, while [2] only considers modifications of constants in *existing* predicates (predicate-level refinements), GOOSEDB also supports *adding* and *removing* predicates (query-level refinements). To this end, GOOSEDB lets the user specify which predicates are eligible for modification. For each of these predicates, it creates variables representing all possible constant values that the predicate can take. For modification of the scoring function, GOOSEDB answers the question of how to get a tuple r into the top- k like [3], by adding a constraint over the sum of indicators, where each indicator represents whether r beats another tuple. A major extension is that, instead of requiring the user to explicitly specify the tuple(s) of interest, GOOSEDB ensures that the required *number of tuples from the corresponding group(s)* appears in the top- k . This is achieved by mapping the constraint to indicators and summing these indicators over all group members in one group. Tuples not selected by the predicates are assigned an indicator value of 0. A sum greater than or equal to c implies that at least c tuples from the group are ranked in the top- k .

3.3 Demonstration Plan

Overview. We will demonstrate the functionality using intuitive scenarios like Example 1.1 (Figure 2) and real-world datasets like *Student*¹ (Figure 1), which contains demographic and grade data of 395 students from Portuguese high schools. The runtime of

¹<https://archive.ics.uci.edu/ml/datasets/student+performance>

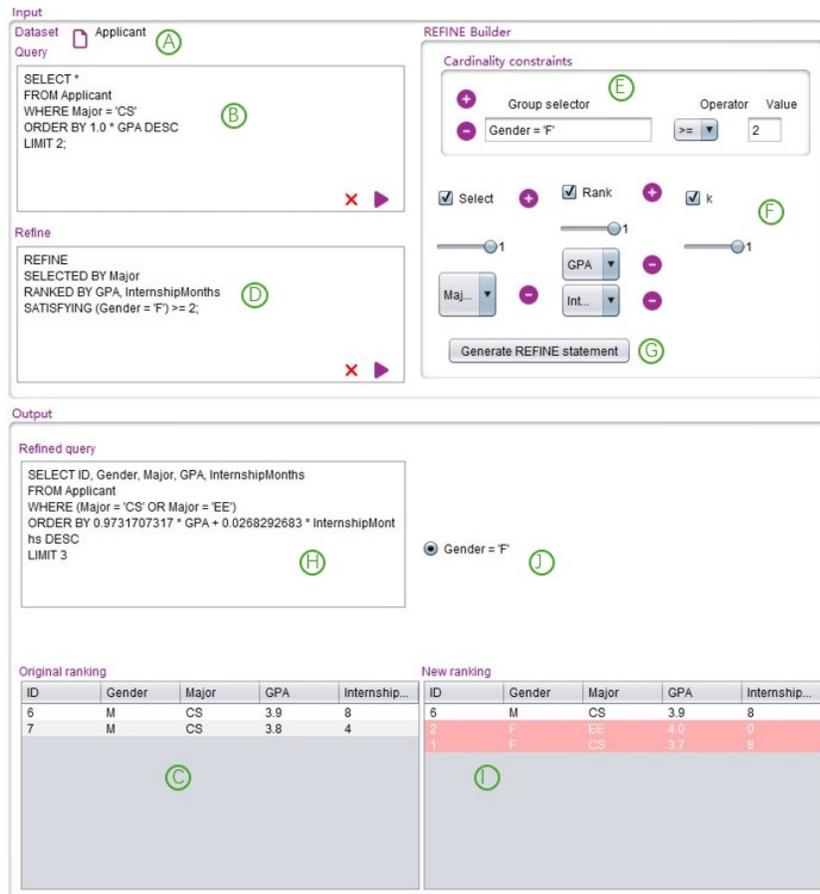


Figure 2: Interface of GooseFACE

GOOSEDB varies depending on data size, query complexity, and constraints. On *Student*, our test refinements complete within 30 seconds. We will also prepare other datasets (sports, university ranking) that are easy to understand and likely to generate interest among conference attendees.

Interaction. In addition to walking the audience through examples like Figures 1 and 2, we let them explore the functionality and flexibility of GOOSEDB and GOOSEFACE interactively by:

- (1) Choosing the interface they prefer;
- (2) Providing or selecting a different dataset;
- (3) Using a different original query;
- (4) Adjusting a cardinality constraint, including changing the group, the constant, and/or the operator;
- (5) Choosing different selection/ranking attributes;
- (6) Configuring user preferences for the optimization goal.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1956096 and was supported in part by NSF grants 2312931 and 2106176.

REFERENCES

- [1] 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.
- [2] Felix S. Campbell, Alon Silberstein, Julia Stoyanovich, and Yuval Moskovitch. 2024. Query Refinement for Diverse Top-k Selection. *Proc. ACM Manag. Data* 3, 3 (2024), 166.
- [3] 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.
- [4] Wesley W. Chu and Qiming Chen. 1994. A Structured Approach for Cooperative Query Answering. *IEEE Transactions on Knowledge and Data Engineering* 6, 5 (1994), 738–749.
- [5] Gurobi Optimization, LLC. 2024. Gurobi Optimizer Reference Manual. <https://www.gurobi.com>
- [6] Jinyang Li, Yuval Moskovitch, Julia Stoyanovich, and H. V. Jagadish. 2023. Query Refinement for Diversity Constraint Satisfaction. *Proc. VLDB Endow.* 17, 2 (2023), 106–118.
- [7] Chaitanya Mishra and Nick Koudas. 2009. Interactive query refinement. In *12th International Conference on Extending Database Technology, EDBT 2009*, Vol. 360. 862–873.
- [8] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD 2019*. 1981–1984.
- [9] Suraj Shetiya, Ian P. Swift, Abolfazl Asudeh, and Gautam Das. 2022. Fairness-Aware Range Queries for Selecting Unbiased Data. In *38th IEEE International Conference on Data Engineering, ICDE 2022*. IEEE, 1423–1436.
- [10] Wenjian Xu, Zhian He, Eric Lo, and Chi-Yin Chow. 2016. Explaining Missing Answers to Top-k SQL Queries. *IEEE Transactions on Knowledge and Data Engineering* 28, 8 (2016), 2071–2085.