# Instance-Optimized String Fingerprints (Extended Abstracts)


Mihail Stoian*
University of Technology Nuremberg
Nuremberg, Germany
mihail.stoian@utn.de

Johannes Thürauf*
University of Technology Nuremberg
Nuremberg, Germany
johannes.thuerauf@utn.de

Andreas Zimmerer
University of Technology Nuremberg
Nuremberg, Germany
andreas.zimmerer@utn.de

Alexander van Renen
University of Technology Nuremberg
Nuremberg, Germany
alexander.van.renen@utn.de

Andreas Kipf
University of Technology Nuremberg
Nuremberg, Germany
andreas.kipf@utn.de



## ABSTRACT

Recent research found that cloud data warehouses are text-heavy. However, their capabilities for efficiently processing string columns remain limited, relying primarily on techniques like dictionary encoding and prefix-based partition pruning.

In recent work, we introduced string fingerprints—a lightweight secondary index structure designed to approximate `LIKE` predicates, albeit with false positives. This approach is particularly compelling for columnar query engines, where fingerprints can help reduce both compute and I/O overhead. We show that string fingerprints can be optimized for specific workloads using mixed-integer optimization, and that they can generalize to unseen table predicates. On an IMDb column evaluated in DuckDB v1.3, this yields table-scan speedups of up to 1.36×.



**VLDB Workshop Reference Format:**
Mihail Stoian, Johannes Thürauf, Andreas Zimmerer, Alexander van Renen, and Andreas Kipf. Instance-Optimized String Fingerprints (Extended Abstracts). VLDB 2025 Workshop: Applied AI for Database Systems and Applications (AIDB 2025).

**VLDB Workshop Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/utndatasystems/string-fingerprints.


## 1 INTRODUCTION

The last three years have shown us that natural language, and implicitly unstructured text, can be *the* language we use to talk to machines [22, 26]. When it comes to the data itself, we can see a similar trend. Indeed, recent research shows that cloud data warehouses are rather text-heavy [28–30]. Yet, how advanced are our techniques to deal with such kind of data? Our work scratches the surface by proposing a lightweight secondary index that boosts queries, i.e., `LIKE` (and simple `REGEX`) predicates, on columns of this data type.

*The author contributed equally to this work.




Proceedings of the VLDB Endowment. ISSN 2150-8097.

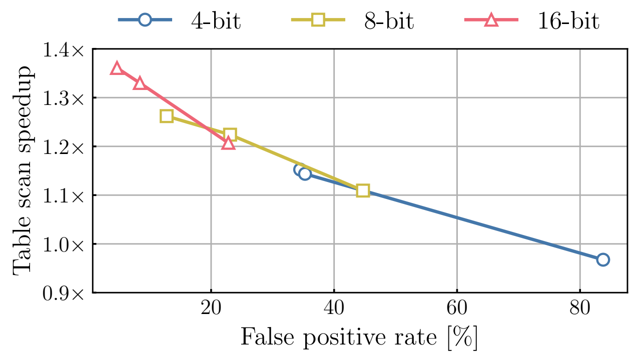




**Figure 1: Using string fingerprints to speedup table scans with `LIKE` predicates on IMDb's `title` column in DuckDB v1.3. Various instance-optimized partitions incur different false positive rates which correlate with the speedup achieved.**

**Motivation.** Transforming a data column to a sparser representation, which can still be queried—with possible allowance of false positives—enables many performance improvements; think of bloom filters [5] and range filters [11, 13, 33], just to name a few from our research field. However, with the exception of a few examples, e.g., compression [7], indexing [21, 27, 33, 34], cardinality estimation [19], it seems that strings are still waiting for more attention, despite their prevalence as data type. Indeed, even recent progress on (updatable) bitmap indices focus on numeric data only [31].

**String Fingerprints.** As a by-product of recent work on robust query processing [25], we introduced string fingerprints as a mechanism that can mimic a `LIKE` predicate, albeit with false positives. The key idea is to compactly represent the set of constituting letters of a string $S$ in a fixed number of bins that can be represented as a binary mask of fixed bitwidth (#bins ≙ bitwidth). For a given pattern $P$, the generic form of a `LIKE` predicate, $S$.contains($P$), evaluates to false if the fingerprint of $P$ is not a *subset* of that of $S$.

**Applications.** This rather concise representation has a twofold role: (a) By attaching a fingerprint-column to the table, we can skip non-qualifying rows, and (b) if we maintain a dictionary with the fingerprint-values seen in the partition, we can use them to skip non-qualifying partitions. For instance, production systems offer the latter for prefix/suffix-based predicates only [35]. In what follows, we focus on the former.

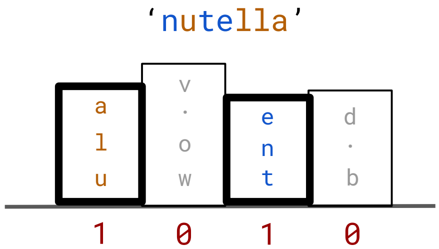



**Figure 2: String fingerprints are bitmasks indexing letter bins**

**Contribution.** Inspired by the recent interest of production systems towards instance-optimized database components [12, 23], we show that string fingerprints can be instance-optimized with respect to the workload and the data itself by using state-of-the-art mixed-integer optimization. In particular, on IMDb's `title` column of the JOB benchmark [20], one can speed up table scans by up to 1.36× when using 16-bit fingerprints, as shown in Figure 1. Moreover, we empirically show that the so instance-optimized fingerprints even generalize to *unseen* table predicates.

**Related Work.** Work on indexing text using $n$-grams dates back to the seminal work of Ogawa and Matsuda [21], and has been further developed—especially in the context of indexing for regular expressions [8, 16, 18, 27, 34]—where the key idea is to select a near-optimal subset of $n$-grams to index (under constraints such as space). With string fingerprints, we argue that, particularly for individual letters (1-grams), there is no need to retain only a subset. Instead, we can index all grams to minimize the false positive rate.

**Overview.** We first introduce the preliminaries of string fingerprints and then present our main technical contribution, an approach to instance-optimize them using mixed-integer optimization (Sec. 2.1). We evaluate the approach on both seen and unseen predicates in Sec. 3. We conclude with future work in Sec. 4.

## 2 INSTANCE-OPTIMIZED FINGERPRINTS

**Preliminaries.** To better understand the intuition behind the MIP formulation in Sec. 2.1, let us next outline how fingerprints work.

The main intuition is that a pattern $P$ has a chance to be contained in a string value $S$, i.e., $S$.`contains`($P$), if the letters of $P$ are a *subset* of that of $S$. To see this, consider the pattern `'%utn%'` and the strings `'nutella'` and `'tone'`. When computing their letter sets, we observe that the first *could* qualify for the pattern, while the latter cannot, since the letter set of `'utn'` is *not* a subset of {`'t'`, `'o'`, `'n'`, `'e'`}. On the other hand, even though the first string value qualifies, it is indeed a false positive. In particular, note that this representation cannot generate false negatives.

**Example.** Instead of indexing all letters, we first partition them. Consider the example in Figure 2 for the string `'nutella'`. The letter space is partitioned in 4 bins (#bins ≜ bitwidth). The letters `'a'`, `'l'`, and `'u'` fall into the first bin (read from left), while the others in the third bin. Hence, its fingerprint reads `1010`. Similarly, we can compute the fingerprints of `'utn'` and `'tone'`, which are `1010` and `0110`, respectively—observe that the letter `'o'` falls into the second bin. In particular, note that the fingerprint of `'utn'` is indeed a subset of that of `'nutella'`, i.e., `1010` ⊆ `1010`, while the same does not hold for `'tone'`, i.e., `0110` ⊈ `1010`. Consequently, the partitioning produced a false positive for `'nutella'` and correctly classified `'tone'` as a true negative.

**Table 1: Notation of the parameters used in the mixed-integer optimization model (1)**

| Parameter | Description |
|---|---|
| $\mathcal{A}$ | Set of characters to be partitioned into the bins |
| $n$ | Number of bins (≜ bitwidth) |
| $\mathcal{W}$ | Words, i.e., set of the column's string values |
| $Q$ | Set of given query patterns |
| $f(\cdot)$ | Function that returns for a query the set of words in $\mathcal{W}$ that contain the query as substring |
| $s_i$ | Denoting the $i$th character of string $s$ |
| $len(s)$ | Denoting the length of string $s$ |
| $[l]$ | For $l \in \mathbb{N}$, it represents the set $\{1, \ldots, l\}$ |

### 2.1 Optimal Partitioning

Mixed-integer linear optimization [9, 10, 32] is a fundamental tool in operations research, which has been successfully applied in different domains including supply chain management [9] and machine learning [4]. Over the last decades enormous computational progress has been made by developing new solution methods and enhancing existing ones leading to efficient optimization solvers such as Gurobi [15], CPLEX [17], and SCIP [6]. Combining the algorithmic advancements with modern computational resources makes it possible to solve optimization problems that were out of scope decades ago.

We now present the developed mixed-integer linear optimization model solving which yields an optimal partitioning. Here, optimal means that for the given query patterns $Q$ and a column's string-values—henceforth, words $\mathcal{W}$—the computed partitioning maximizes the number of correctly classified data, which is equivalent to minimizing the false positive rate. In particular, if the optimization model is solved to global optimality, the used mathematical methods guarantee that the resulting partitioning has the lowest possible false positive rate for the given queries and data.

**Notation.** We introduce the necessary notation in Table 1. In addition, we use the following optimization variables. For each bin $i \in [n]$ and character $a \in \mathcal{A}$, the binary variable $x_{a,i} \in \{0, 1\}$ evaluates to 1 if character $a$ is in bin $i$. Otherwise, $x_{a,i} = 0$ holds. Thus, the $x$ variables determine the optimized partitioning of the characters into the bins. For each string $s \in Q \cup \mathcal{W}$ and bin $j$, the binary variable $d_j^s \in \{0, 1\}$ indicates if there is a letter of string $s$ that is in bin $j$, i.e., $d_j^s = 1$ holds, and if this is not the case, $d_j^s = 0$ is satisfied. Consequently, $d_j^s, j \in [n]$, represent the string fingerprint of string $s$. For each query $q \in Q$ and word $w \in \mathcal{W} \setminus f(q)$, the variable $\eta^{w,q} \in \{0, 1\}$ evaluates to one if the optimized partitioning correctly determines that query $q$ is not contained in word $w$, i.e., the optimized partitioning does not produce a false positive for the considered query $q$ and word $w$. However, if the optimized partitioning produces a false positive for this query and word, then $\eta^{w,q} = 0$ holds.

**Model.** The aim of the optimization model is to maximize the number of correctly classified query-word combinations, i.e., to maximize $\sum_{q\in\mathcal{Q}}\sum_{w\in\mathcal{W}\setminus f(q)}\eta^{w,q}$. We note that this is equivalent to minimizing the number of falsely classified query-string combinations. Using the introduced variables and notation, the optimization model to compute an optimal partitioning w.r.t. the given query-word combinations is given by

$$\max_{x,d,\eta} \quad \sum_{q\in\mathcal{Q}}\sum_{w\in\mathcal{W}\setminus f(q)}\eta^{w,q} \tag{1a}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} x_{a,j} = 1, \quad a\in\mathcal{A}, \tag{1b}$$

$$x_{s_i,j} \le d_j^s, \quad i\in[\mathrm{len}(s)],\ j\in[n],\ s\in\mathcal{Q}\cup\mathcal{W}, \tag{1c}$$

$$d_j^s \le \sum_{i\in[\mathrm{len}(s)]} x_{s_i,j}, \quad j\in[n],\ s\in\mathcal{Q}\cup\mathcal{W}, \tag{1d}$$

$$d_j^q \le d_j^w, \quad j\in[n],\ q\in\mathcal{Q},\ w\in f(q), \tag{1e}$$

$$\eta^{w,q} \le \sum_{j=1}^{n}(1-d_j^w)d_j^q, \quad w\in\mathcal{W}\setminus f(q),\ q\in\mathcal{Q}, \tag{1f}$$

$$x\in\{0,1\}^{|\mathcal{A}|\times n},\ d\in\{0,1\}^{(|\mathcal{W}|+|\mathcal{Q}|)\times n}, \quad \eta\in\{0,1\}^{|\mathcal{W}|\times|\mathcal{Q}|-\sum_{q\in\mathcal{Q}}|f(q)|}. \tag{1g}$$

**Model Description.** In Constraints (1b), we ensure that each character of $\mathcal{A}$ is exactly assigned to one bin. By Constraints (1c) and (1d), for each string $s\in\mathcal{Q}\cup\mathcal{W}$ we determine the string fingerprint $d^s$ w.r.t. the chosen partitioning given by $x$. More precisely, $d_j^s$ equals one if and only if string $s$ contains at least one letter that is in bin $j$. Otherwise, it is zero. Constraints (1e) ensure that we do not have any false negatives by enforcing that every string that contains the query has a string fingerprint that includes the one of the query. We now consider a query $q\in\mathcal{Q}$ and word $w\in\mathcal{W}$ that does not contain query $q$, i.e., $w\in\mathcal{W}\setminus f(q)$. If $d_j^q\le d_j^w$ for all $j\in[n]$ holds, then we have wrongly classified the word $w$ to contain $q$. Further, if $d_j^q\le d_j^w$ is satisfied for all $j\in[n]$, the right-hand side of Constraint (1f) evaluates to zero due to $d_j^q, d_j^w\in\{0,1\}$. Consequently, in this case Constraint (1f) implies $\eta^{w,q}=0$ and we cannot increase the objective function by wrongly classified strings. However, if $d_j^q\le d_j^w$ does not hold, i.e., we correctly determine that word $w$ does not contain $q$, then the right-hand side of Constraint (1f) is at least one. Consequently, we can set $\eta^{w,q}=1$ and increase the objective function by correctly identifying that word $w$ does not contain $q$. The objective function (1a) then maximize the number of correctly classified strings that do not contain the corresponding query, which is equivalent to minimizing the number of false positives. Consequently, solving the presented optimization model to global optimality leads to an optimized partitioning that minimizes the false positive rate w.r.t. the given queries and data.

We note that Model (1) is a mixed-integer nonlinear optimization problem in the presented form due to the products of binary variables in Constraints (1f). However, using standard techniques of mixed-integer optimization, these products of binary variables can be equivalently reformulated with the help of additional variables and linear constraints, e.g., see [14]. This leads to a mixed-integer linear optimization model, which we used in our computational study. We further neglect Constraints (1e) in our computations because they are mathematically redundant, i.e., they can be removed without changing the set of optimal solutions. This directly follows from the fact that for $d_j^q=1$ there exists a character in bin $j$ that is included in query $q$ due to Constraints (1c) and (1d). However, this character is also included in word $w$ due to $q\in\mathcal{Q}$ and $w\in f(q)$. Consequently, Constraints (1c) implies $d_j^w=1$ and then the corresponding Constraint (1e) is satisfied. The latter is directly valid for the case $d_j^q=0$. This formal discussion proves that the computed partitioning and corresponding string fingerprints cannot produce false negatives.

Solving Model (1) to global optimality using state-of-the-art solvers computes a partitioning that is optimal w.r.t. the considered queries and data, i.e., it has the lowest possible false positive rate. However, for a large number of queries, words, and bins, the number of variables and constraints is enormous, which makes it challenging to solve these models to global optimality in a reasonable time. To ensure that the computational time stays within practical limits, a time limit can be imposed on the optimization process. As shown in the evaluation, this leads to computing a high-quality partitioning, that may not be optimal, but efficiently minimizes the false positive rate within the time limit. Moreover, the following computational results show that the optimized partitioning also performs well for unseen queries and data.

# 3 EVALUATION

**Setup.** We run the queries single-threaded on a single node Intel® Xeon® Gold 5318Y CPU (24 cores, 48 hyper-threads). The machine is equipped with 128GB DDR4 main memory and runs Ubuntu 24.04. We use DuckDB v1.3.0 as query engine. The MIP solutions are computed with Gurobi 12.0.1 ([15]), with a time limit of 300 s and a thread limit of 48. In the following, we denote as optimization time the total time required for reading the data, building the optimization model, and solving it.

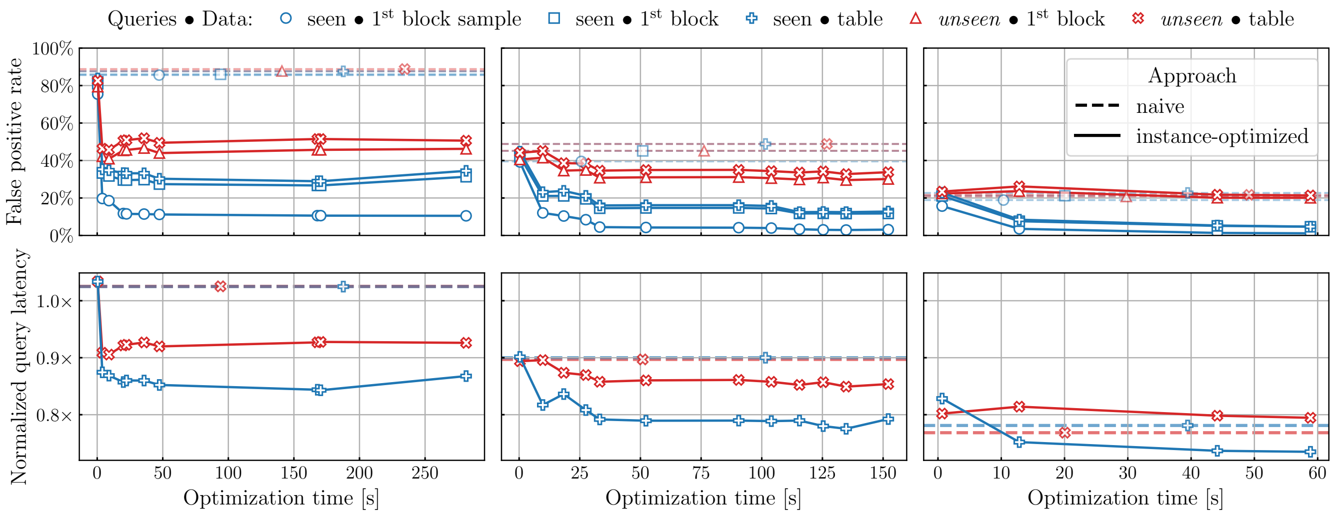
**Benchmark.** We consider the real-world IMDb dataset from the JOB benchmark [20] and take as reference its `title` column with 2.53 M movie names; since we currently optimize for printable bytes only (100 distinct ones, i.e., $|\mathcal{A}|=100$), the table is reduced to 2.37 M tuples. We compose a 300-query workload consisting of the 10 highest-, mid-, and lowest-frequency $k$-grams for each $k\in\{1,\ldots,10\}$ extracted from the column. These are randomly split into 20 seen and 280 unseen queries.

Due to DuckDB v1.3.0's limited support for complex predicate pushdown, we simulate the pushdown of our bitmask check by (a) measuring the time to perform the bitmasked table scan, (b) building an auxiliary column that indicates the result, and (c) and measuring the time of the new query. The reported time is (a) + (c).

## 3.1 Table Scan Speedup

One of the main applications of string fingerprints is accelerating table scans. Namely, in the context of a columnar query engine, one can attach the fingerprint column and, at query time, evaluate the corresponding predicate first:

```
where [...] title_fp & pattern_mask = pattern_mask
and title like '%{pattern}%' [...].
```




**(a) 4-bit** **(b) 8-bit** **(c) 16-bit**

**Figure 3: Effect of string fingerprints on table scans over IMDb's `title` column for various fingerprint bitwidths $\in \{4, 8, 16\}$ and (query, data) pairs. The 300-query workload consists of the 10 highest-, mid-, and lowest-frequency column's $k$-grams for each $k \in \{1, \ldots, 10\}$. Queries are split into 20 *seen* and 280 *unseen* patterns. Instance-optimized partitions are trained on a 50-tuple sample from the first data block using the seen queries. We also plot a subset of the intermediate solutions obtained by the solver during the optimization process. The naive, workload-agnostic baseline assigns characters in a round-robin manner.**

In this case, the `LIKE` predicate may need to be evaluated on significantly fewer tuples. This effect is visualized in Figure 3, where, we show the false positive rates (FPRs) for various fingerprints bitwidths $\in \{4, 8, 16\}$ (upper subplots) and the query latencies for seen and unseen queries on the full table (lower subplots).

1$^{\text{st}}$ **Observation.** Despite being trained on a random 50-tuple sample of the first data block (=$2^{16}$ tuples) of the table, we observe that the FPRs of the instance-optimized fingerprints remain competitive even on the full table. This is also reflected in the latency numbers, where in the 16-bit setting, the speedup reaches 1.36×. The same holds for the unseen queries, for which the attained speedup reads 1.26×. The optimization time spent for this setting amortizes for the unseen queries already in the 4th run of the workload.

2$^{\text{nd}}$ **Observation.** Given the fact that the pattern lengths are bounded above by 10, the instance-optimized setting is not worth it with increasing bitwidth. This is due to the fact that with a larger bitwidth (and a rather bounded alphabet), the bin densities are much lower, thus allowing for sparser fingerprints, both for patterns and column string values.

3$^{\text{rd}}$ **Observation.** For the 4-bit and 8-bit setting, the underlying optimization problems to compute the partitions in Figure 3 cannot be solved to global optimality within 300 s. However, we obtain a MIP optimality gap of around 5.6% and 1.9%, respectively. Consequently, the quality of the current best partition is rather close to the one of an optimal solution (w.r.t. the given query-data). In particular, optimizing for at most 60 s is sufficient to produce solutions that are competitive with the fastest in terms of query performance; for the 8-bit case, the gap reads 1.82%.

# 4 CONCLUSION & FUTURE WORK

String fingerprints are a promising research direction towards efficiently querying string columns. The key idea is to partition the letter space such that the false positive rate on a given workload is minimized. More importantly, unlike recent caching mechanisms [12, 23], string fingerprints (a) can be both used to instance-optimize the database and (b) generalize to *unseen* queries. Regardless of the extent of predicate pushdown in the system, instance-optimized string fingerprints act as table scan accelerators by reducing the number of actual string predicate evaluations.

**Fingerprints $\bowtie$ *N*-grams.** To achieve even better FPRs, one could consider using larger grams instead of letters (1-grams) only. The presented MIP formulation (Sec. 2.1) can be applied to this setting with adaptations, however, the corresponding optimization problems become more challenging since the alphabet ($\mathcal{A}$; Tab. 1), and hence the number of optimization variables, naturally increase. Using larger grams fits the line of research on $n$-gram indexing for regular expressions [8, 16, 18, 21, 27, 34], which instead optimizes which subset of $n$-grams to index.

In future work, we also plan to investigate how string fingerprints can be used for (a) string zonemaps—particularly for large bitwidths, (b) `LIKE` cardinality estimation [1–3, 19, 24], and (c) clustering, even across multiple string columns.

**Acknowledgments.** The authors gratefully acknowledge the scientific support and HPC resources provided by the Erlangen National High Performance Computing Center (NHR@FAU) of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). The hardware is funded by the German Research Foundation (DFG).

The authors acknowledge the use of DeepL and OpenAI's ChatGPT for partly editing and polishing the text and figures for spelling, grammar, and stylistic improvements. Additionally, ChatGPT was utilized for support in basic coding tasks.

## REFERENCES


[1] Mehmet Aytimur and Ali Cakmak. 2018. Estimating the selectivity of LIKE queries using pattern-based histograms. *Turkish J. Electr. Eng. Comput. Sci.* 26, 6 (2018), 3320–3335. https://doi.org/10.3906/ELK-1806-96

[2] Mehmet Aytimur and Ali Cakmak. 2021. Using positional sequence patterns to estimate the selectivity of SQL LIKE queries. *Expert Syst. Appl.* 165 (2021), 113762. https://doi.org/10.1016/J.ESWA.2020.113762

[3] Mehmet Aytimur, Silvan Reiner, Leonard Wörteler, Theodoros Chondrogiannis, and Michael Grossniklaus. 2024. LPLM: A Neural Language Model for Cardinality Estimation of LIKE-Queries. *Proc. ACM Manag. Data* 2, 1 (2024), 54:1–54:25. https://doi.org/10.1145/3639309

[4] Dimitris Bertsimas, Jack Dunn, Colin Pawlowski, and Ying Daisy Zhuo. 2019. Robust classification. *INFORMS Journal on Optimization* 1, 1 (2019), 2–34. https://doi.org/10.1287/ijoo.2018.0001

[5] Burton H Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[6] Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Liding Xu. 2024. The SCIP Optimization Suite 9.0. arXiv:2402.17702 [math.OC] https://arxiv.org/abs/2402.17702

[7] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661.

[8] Junghoo Cho and Sridhar Rajagopalan. 2002. A fast regular expression indexing engine. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 419–430.

[9] Fran¸cois Clautiaux and Ivana Ljubić. 2025. Last fifty years of integer linear programming: A focus on recent practical advances. *European Journal of Operational Research* 324, 3 (2025), 707–731. https://doi.org/10.1016/j.ejor.2024.11.018

[10] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. 2014. *Integer Programming*. Graduate Texts in Mathematics, Vol. 271. Springer, Cham. xii+456 pages. https://doi.org/10.1007/978-3-319-11008-0

[11] Niv Dayan, Ioana O. Bercea, Pedro Reviriego, and Rasmus Pagh. 2023. InfiniFilter: Expanding Filters to Infinity and Beyond. *Proc. ACM Manag. Data* 1, 2 (2023), 140:1–140:27. https://doi.org/10.1145/3589285

[12] Jialin Ding, Matt Abrams, Sanghita Bandyopadhyay, Luciano Di Palma, Yanzhu Ji, Davide Pagano, Gopal Paliwal, Panos Parchas, Pascal Pfeil, Orestis Polychroniou, Gaurav Saxena, Aamer Shah, Amina Voloder, Sherry Xiao, Davis Zhang, and Tim Kraska. 2024. Automated Multidimensional Data Layouts in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 55–67. https://doi.org/10.1145/3626246.3653379

[13] Navid Eslami and Niv Dayan. 2024. Memento Filter: A Fast, Dynamic, and Robust Range Filter. *Proc. ACM Manag. Data* 2, 6 (2024), 244:1–244:27. https://doi.org/10.1145/3698820

[14] Fred Glover and Eugene Woolsey. 1974. Converting the 0-1 Polynomial Programming Problem to a 0-1 Linear Program. *Operations Research* 22, 1 (1974), 180–182.

[15] Gurobi Optimization, LLC. 2024. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[16] Bijit Hore, Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. 2004. Indexing text data under space constraints. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management*. 198–207.

[17] IBM ILOG CPLEX Optimizer. [n.d.]. IBM ILOG CPLEX Optimizer. https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer

[18] Younghoon Kim, Hyoungmin Park, Kyuseok Shim, and Kyoung-Gu Woo. 2013. Efficient processing of substring match queries with inverted variable-length gram indexes. *Information Sciences* 244 (2013), 119–141.

[19] Suyong Kwon, Kyuseok Shim, and Woohwan Jung. 2025. Cardinality Estimation of LIKE Predicate Queries using Deep Learning. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[20] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[21] Yasushi Ogawa and Toru Matsuda. 1998. Optimizing query evaluation in n-gram indexing. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 367–368.

[22] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744.

[23] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 347–359. https://doi.org/10.1145/3626246.3653395

[24] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. 2020. Astrid: Accurate Selectivity Estimation for String Predicates using Deep Learning. *Proc. VLDB Endow.* 14, 4 (2020), 471–484. https://doi.org/10.14778/3436905.3436907

[25] Mihail Stoian, Andreas Zimmerer, Skander Krid, Amadou Latyr Ngom, Jialin Ding, Tim Kraska, and Andreas Kipf. 2025. Parachute: Single-Pass Bi-Directional Information Passing. arXiv:2506.13670 [cs.DB] https://arxiv.org/abs/2506.13670

[26] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971* (2023).

[27] Dominic Tsang and Sanjay Chawla. 2011. A robust index for regular expression queries. In *Proceedings of the 20th ACM international conference on Information and knowledge management*. 2365–2368.

[28] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3694–3706.

[29] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. In *VLDB*.

[30] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest*.

[31] Junchang Wang and Manos Athanassoulis. 2024. CUBIT: Concurrent Updatable Bitmap Indexing. *Proceedings of the VLDB Endowment* 18, 2 (2024), 399–412.

[32] Laurence A. Wolsey. 2020. *Integer Programming*. John Wiley & Sons, Inc. https://doi.org/10.1002/9781119606475

[33] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 2018 International Conference on Management of Data*. 323–336.

[34] Ling Zhang, Shaleen Deep, Jignesh M. Patel, and Karthikeyan Sankaralingam. 2025. An Evaluation of N-Gram Selection Strategies for Regular Expression Indexing in Contemporary Text Analysis Tasks. arXiv:2504.12251 [cs.DB] https://arxiv.org/abs/2504.12251

[35] Andreas Zimmerer, Damien Dam, Jan Kossmann, Juliane Waack, Ismail Oukid, and Andreas Kipf. 2025. Pruning in Snowflake: Working Smarter, Not Harder. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 757–770. https://doi.org/10.1145/3722212.3724447