Extensible and Robust Evaluation of Similarity Queries

Daniel Schmitt
University of Salzburg, Austria
danielulrich.schmitt@plus.ac.at

Thomas Hütter
Software Competence
Center Hagenberg, Austria
thomas.huetter@scch.at

Nikolaus Augsten
University of Salzburg, Austria
nikolaus.augsten@plus.ac.at

ABSTRACT

We study the similarity join problem from a systems perspective. A similarity join retrieves all similar record pairs from two collections based on a given distance function. Existing solutions are often optimized for a single distance function and domain. Such monolithic solutions are limited in both their extensibility to new distance functions and their robustness against changing data characteristics.

To address these challenges, we introduce FAST, a similarity join algorithm designed for extensible and robust query evaluation. It leverages a novel abstraction called reductions, which transform similarity join problems from complex domains into simpler ones. A reduction graph is constructed to systematically enumerate query plans. Since cost models for similarity queries are typically unavailable, FAST employs runtime partitioning and a sampling-based strategy to select a near-optimal query plan with performance guarantees. It can utilize prebuilt indexes or build them on-the-fly, incorporating caching techniques to accelerate index construction and probing. Extensive experiments across diverse datasets, domains, and distance functions show that FAST consistently performs close to the optimal plan. Finally, two case studies highlight its strength as a baseline and its utility for prototyping future similarity join algorithms.

PVLDB Reference Format:
Daniel Schmitt, Thomas Hütter, and Nikolaus Augsten. Extensible and Robust Evaluation of Similarity Queries. PVLDB, 18(11): 3868 - 3882, 2025.
doi:10.14778/3749646.3749660

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DatabaseGroup/fast-similarity-evaluation.

1 INTRODUCTION

Similarity joins are a fundamental operation in data integration, entity resolution, and information retrieval [11, 12, 16, 42, 54, 67, 79]. They use a predefined *distance function* to retrieve all pairs of similar records from two datasets. A common variant, the *range join*, matches records if their distance is within a threshold. The choice of distance function is application-specific and heavily depends on the *domain* of the records: for sets, for example, Jaccard or overlap similarity is common [47]; string joins may rely on edit distance [30] or on Jaro similarity [17]; and for hierarchical data like JSON or XML, tree-based distances such as the JSON edit distance [28] or pq-grams [5] have been used.

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749660

Despite decades of progress in developing efficient physical operators for specific distance functions [19, 26, 27, 29, 35, 44, 57, 71, 79, 80, 83], integrating similarity joins into database systems remains challenging. In particular, two major system-level challenges persist [3]:

- *Extensibility:* Similarity measures are highly application-dependent, and numerous distance functions have been proposed [17, 49, 51]. A system must allow new distance functions to be added with low engineering effort.
- *Robustness*: The performance of similarity join algorithms varies drastically depending on dataset characteristics and query parameters. Effective operator selection is essential for robust system performance.

As an example of the *extensibility* challenge, consider a system that implements a physical operator for set similarity joins. Extending such a system to support tree similarity joins (e.g., using tree edit distance for XML documents) typically requires substantial engineering effort: a suitable join algorithm must be selected or developed, specialized index structures (e.g., pq-grams [5] or positional label sets [29]) must be implemented, and new access and join logic must be written. The existing code for set similarity joins is not easily reusable.

Building a system that is *robust* across diverse workloads is challenging: the performance of different similarity join algorithms varies significantly depending on dataset characteristics (e.g., data distribution, record sizes) and query parameters (e.g., the similarity threshold) [30, 57]. Experimental evaluations consistently show that no single algorithm performs best across all scenarios; a poor algorithm choice may degrade performance by orders of magnitude [57]. The lack of suitable cost models [3] complicates query optimization.

We argue that these challenges arise from how similarity join algorithms are typically designed: as monolithic, tightly coupled implementations. Most approaches extract features from input records and evaluate a pairwise *filter condition*, for example, extracting *pq*-grams from trees. A *physical join operator* then uses this filter to identify matching record pairs, such as via an index-nested loop join with prefix filtering. Current solutions hardcode both the filter condition and the physical operator into a single, inseparable unit. This monolithic design has serious drawbacks: it limits robustness, as fixed internal plans cannot adapt to varying workloads, and it hinders extensibility, as components are difficult to reuse. As a result, developers are forced to implement and maintain a growing collection of isolated, specialized algorithms.

Modularization using Reductions. We propose addressing these challenges through a modular design centered around a new abstraction: *reductions*. A reduction explicitly defines how a similarity join problem in one domain (e.g., trees with tree edit distance) can be transformed into a similarity join in another, typically simpler domain (e.g., sets with structural similarity). While such transformations have appeared *implicitly* in prior work, they were tightly coupled to specific algorithms and not reusable across contexts [22, 27, 29, 77].

By elevating reductions to first-class, formally defined components, we decouple the specification of similarity from its physical execution. This enables a clean separation of concerns: domain experts define reductions tailored to their data and distance functions, while system developers build and optimize a shared set of physical join operators over well-supported domains (e.g., sets, strings, or trees). The lack of a clear separation between reduction and execution has repeatedly led to suboptimal algorithms, where effective reductions were combined with inefficient join operators [27, 29, 50].

This modular architecture offers two key benefits. First, it improves *extensibility*: new distance functions can be supported by defining a reduction to an existing domain, without implementing a full join algorithm from scratch. Second, it enhances *robustness*: reductions can be composed with different physical operators, enabling the system to explore a larger space of query plans.

To systematically capture these combinations, we introduce the notion of a *reduction graph*, where nodes represent intermediate domains and edges correspond to reductions or physical join operators. This abstraction allows us to enumerate a wide variety of evaluation strategiesmany of which have not appeared in previous workand choose among them adaptively at runtime.

Query Planning. Using the reduction graph, we enumerate a set of query plans. Traditional database systems rely on cost models to optimize query plans, but such models are not available for similarity join operators [3]. Since the performance of these operators depends heavily on dataset characteristics and query parameters, selecting an efficient plan is challenging. This problem is related to estimating the cardinality of similarity joins, which has applications in predicate and join ordering [31, 48, 61, 75]. However, existing estimators only predict the output size of a similarity join, whereas the performance of a similarity join often depends on the size of intermediate results; information that is not captured by these estimators.

To address the lack of suitable cost models, we propose a sampling-based strategy: each plan is executed for a short timeslice, and its progress guides future choices. Plan selection is modeled as a Markov decision process, and the *Upper Confidence Tree* (UCT) algorithm [41] guarantees eventual convergence to the best query plan.

In summary, our contributions are as follows:

- Our Fast (**F**ully **A**utomatic **S**imilarity **T**ransformation) algorithm leverages the novel abstraction of a *reduction graph* to enumerate all possible evaluation plans.
- For *query planning*, we propose a feedback strategy based on runtime partitioning and provide performance guarantees w.r.t. the optimal plan. We do not require cost models.
- To reduce the overhead of partial indexes, we propose a *caching strategy* that stores intermediate results across repeated selections of the same evaluation plan.
- Extensive experiments on twenty datasets, three domains, and various distances show the robustness of our approach: On average, Fast is only 12% slower than the optimal plan.
- Two case studies show the *extensibility* of Fast and its value as a *benchmark*: Although the fastest query plans rely on techniques that have been known for over a decade, three out of four recent algorithms fail to significantly improve over Fast.

Omitted proofs appear in an extended version of this paper [56].

2 BACKGROUND AND PROBLEM STATEMENT

Similarity Search. Similarity search matches similar record pairs (r,s) from datasets R,S in a domain $\mathcal{T}$. Record similarity is assessed by a distance function $d\colon \mathcal{T}^2 \to \mathbb{R}_0^+$. $\mathcal{D}$ denotes the set of distance functions. We focus on *range similarity join* queries: Given datasets $R,S \subseteq \mathcal{T}$ and threshold ε, the range similarity join $R \tilde{\bowtie}_d^{\varepsilon} S$ finds all pairs $(r,s) \in R \times S$ with $d(r,s) \leq \varepsilon$. In this paper, we focus on self joins ($R = S$) that are common for applications such as entity resolution [79]. However, all results generalize to non-self joins.

Domains and Distance Functions. We focus on sets, strings, and trees as they are widely used in similarity search [4]. In particular, these domains form a ladder of increasing structure: Every (finite) set can be represented as a string and every string can be represented as a tree without losing information. Many algorithms for similarity search take the opposite direction: A reduction in structure allows for faster computations, e.g., by reducing the tree edit distance with cubic runtime to the quadratic time string edit distance [43].

Commonly studied distance functions on strings and trees are their respective edit distances. For sets, we consider all *symmetric* [15] similarity measures. Symmetric similarity measures on sets can be rewritten as a minimum required overlap given the sizes of two sets. Symmetric similarity measures include common similarity measures like Jaccard, Cosine, and Dice, or the Hamming distance.

Upper Confidence Tree (UCT). We leverage UCT [41] for plan selection. UCT finds near-optimal solutions for a *Markov Decision Process* (MDP). For our purposes, an MDP is a tuple $(\mathcal{S},\mathcal{A},\delta,\mathcal{R})$ consisting of a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, a deterministic transition function $\delta\colon \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, and a stochastic reward function $\mathcal{R}$ assigning probability distributions for rewards in $[0,1]$ to states.

UCT [41] balances the aspects of exploration – exploring parts of the tree with little information to aid in future decisions – and exploitation – repeating choices that led to large rewards in the past. To this end, UCT maintains both the average reward r and the number of visits v for each node. To select a path, UCT starts in the root of the UCT tree and always selects child c with parent p maximizing the average reward r_c plus a bias term $w\sqrt{\log(v_p)/v_c}$ with weight w. With increasing visits to c, the certainty increases and the bias decreases. Theoretical guarantees require setting w proportional to the height of the tree [41], but empirically, application-specific tuning of w can lead to improved performance [21]. After achieving a reward at a leaf node, the statistics of all nodes along the path are updated.

The performance of UCT and similar algorithms is measured in the *regret* of their decisions. Regret is the difference in rewards between the selected policy of the learning algorithm and the optimal policy. Our approach requires the notion of *cumulative regret*, which measures the difference in expected maximal rewards compared to the achieved rewards *over all choices* of the algorithm. For UCT, the cumulative regret after n selections is $O(\log n)$ (Theorem 6 in [41]). While other algorithms converge faster than UCT [24, 25], their different notion of regret is not suitable for our purpose. UCT and its variations have been successfully used to solve game [40, 60, 64, 73, 81] and planning [38] problems. Most notably, the strongest Go programs are based on UCT [60] and solve MDPs with branching factors of $\geq$40 and depths $\geq$20 [81]. Our use case is currently significantly smaller in branching factor ($\leq$4) and depth ($\leq$4), leaving room for future extension.

Problem Statement. The goal of this work is to develop a system that is capable of computing similarity joins on sets, strings, and trees with edit distance and symmetric set similarities. Furthermore, the system should satisfy the following properties: (1) **Extensibility:** Support for new distance functions can be added without significant rework. (2) **Robustness:** The system adapts to a given query, e.g., its characteristics, and computes its result efficiently.

3 REDUCE-FILTER-VERIFY FRAMEWORK

Most algorithms for similarity search are based on the *filter-verify* [29, 44, 46, 55] framework. We extend this framework by an explicit *reduce* step, which was an implicit part in the filter step of previous algorithms. A *reduce-filter-verify* algorithm consists of three steps:

Reduce: The dataset, distance function, and threshold are reduced by a *reduction.* Let $\mathcal{D}$ be a set of distance functions. A reduction from domain $\mathcal{T}_1$ to $\mathcal{T}_2$ (where possibly $T_1 = T_2$) is a (partial) function $\rho\colon \mathcal{T}_1 \times \mathcal{D} \times \mathbb{R}_0^+ \to \mathcal{T}_2 \times \mathcal{D} \times \mathbb{R}_0^+$. It maps a record from $\mathcal{T}_1$, a distance function on $\mathcal{T}_1$, and a threshold onto a record from $\mathcal{T}_2$, a distance function on $\mathcal{T}_2$, and a new threshold. Typically, domain $\mathcal{T}_2$ has less structure than $\mathcal{T}_1$; examples include converting trees to sets [6, 29], strings [32, 43, 80], or vectors [43]. We simply write $\rho(r)$, $\rho(d)$, and $\rho(\varepsilon)$ if the other arguments are clear from context; $\rho(R)$ denotes the dataset computed by pointwise application of ρ to the elements in the dataset R. A reduction serves as a filter condition: Pairs removed in the join computed in the reduced domain (computed on $\rho(R)$ and $\rho(S)$ using distance function $\rho(d)$ and threshold $\rho(\varepsilon)$) are guaranteed to *not* be in the result. We refer to this property as *similarity supersets*:

$$R \tilde{\bowtie}_d^{\varepsilon} S \subseteq \rho(R) \tilde{\bowtie}_{\rho(d)}^{\rho(\varepsilon)} \rho(S)$$

Filter: The filtering step is implemented by a physical join operator (or *join operator* for short) that computes the similarity join after reduction: $\rho(R) \tilde{\bowtie}_{\rho(d)}^{\rho(\varepsilon)} \rho(S)$. By the similarity supersets property, all pruned pairs are true negatives, i.e., they are not part of $R \tilde{\bowtie}_d^{\varepsilon} S$. We assume that join operators are index-based: Given a query $\rho(q)$, the index returns a set of candidates including all truly similar pairs. We will state further assumptions on the index structure in Sections 5 and 6. All current state-of-the-art algorithms for similarity search on discrete domains satisfy these assumptions [10, 20, 26, 29, 35, 44, 53, 57, 83].

Verify: The candidates produced by the filtering step include all true positives, but also false positives. The false positives are removed by computing the similarity on the original domain and comparing against the threshold. As computing the similarity on the reduced data is typically faster than on the original data, candidates are often verified on the reduced data first.

4 QUERY PLAN ENUMERATION

Any algorithm following the reduce-filter-verify framework can be decomposed into three parts: reduce, filter, and verify. Hence, any such algorithm can be seen as a query plan composing a specific reduction step, a join operator, and a verification procedure. We introduce the *reduction graph* to enumerate all valid query plans for a given query. Informally, a reduction graph connects distance functions using reductions. Distance functions connected to some algorithm can be evaluated by said algorithm without further reduction.

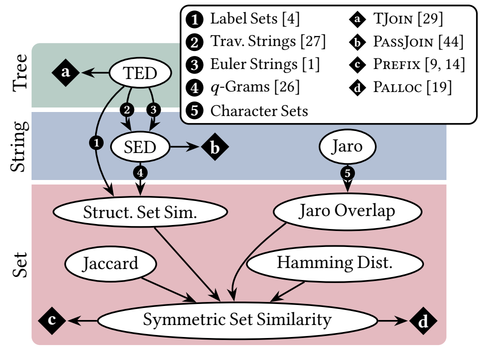


Figure 1: Reduction graph for sets, strings, and trees.

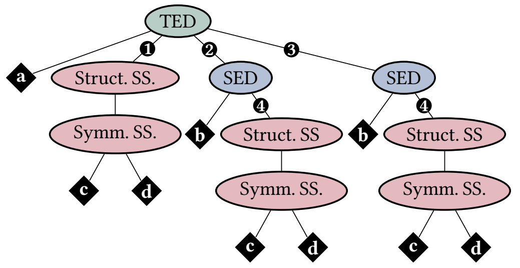


Figure 2: Query plans/MDP for TED represented as a tree.

Definition 4.1 (Reduction Graph). A reduction graph is a directed acyclic edge-labeled multigraph. It distinguishes two types of nodes: (1) A distance node corresponds to a distance function on some domain. (2) Algorithm nodes correspond to a concrete join algorithm. A distance node connects to an algorithm node if a similarity join with that distance function can be evaluated by the algorithm. Edges between distance nodes correspond to reductions between the distance functions and their respective domains.

Paths in the reduction graph starting at some distance node for some distance function d and ending in an algorithm a correspond to query plans of a reduce-filter-verify algorithm: The reduction step is the composition of all reduction edges on the path. The filter step is implemented by the join operator a. For the verification step, we reverse the path and evaluate the reduced distance function for each distance node, pruning dissimilar pairs in the process. To enumerate all possible query plans $\mathcal{P}$ for a given distance function, we perform standard depth-first search from the corresponding distance node. The correctness of such query plans follows from compositionality of reduction functions and the correctness of filtering algorithms.

Example 4.2. Consider the reduction graph depicted in Figure 1. There are nine possible query plans for a similarity query using the *tree edit distance* (TED). Figure 2 visualizes all plans as a tree.

- **Trees:** Use TJoin [29] without any reduction.
- **Sets:** Reduce the trees to sets of node labels and use either Prefix [9, 14] or Palloc [19].
- **Strings:** Use traversal strings [27] or Euler strings [1] to convert the trees to strings. Then, use the *string edit distance*

(SED) algorithm PassJoin [44]. Alternatively, perform another reduction to sets using q-grams [26]. In the latter case, filtering is performed using Prefix [9, 14] or Palloc [19].

5 STATIC INDICES QUERY EVALUATION

Given a query, the reduction graph enumerates all query plans $\mathcal{P}$ to evaluate the query. Similarity join algorithms generally do not have cost models that could be leveraged to select the best plan. Instead, we propose to use a sampling-based approach that executes one query plan for a short duration, measures its performance, and bases its future selections on the previous performance of a plan. The selections based on previous performance use the Upper Confidence Tree (UCT) [41] algorithm to ensure eventual convergence to the fastest plan. In this section, we define requirements on the similarity join algorithm implementing the filtering step, briefly describe the setup and challenges of using UCT, and show how near-optimal query plans can be selected.

5.1 Requirements on Similarity Join Algorithms

As stated in Section 2, we require that all join operators implementing the *filter* step in the *reduce-filter-verify* framework are index-based. The algorithm must satisfy the following requirements: (1) **Independent-Index:** A dataset can be indexed in advance without knowledge about the future queries. (2) **Total-Recall:** For a given query record, the algorithm returns a superset of all similar records in the indexed dataset, possibly including false positives. (3) **Prefix-Search:** The index allows queries on a *prefix* of the dataset. In other words, for a given dataset $R = \{r_1, \ldots, r_n\}$, the algorithm can perform queries on any $R' = \{r_1, \ldots, r_i\}$ with $i \leq n$. Inverted list indexes support such queries by storing the records sorted by record ID.

5.2 Leveraging UCT for Plan Selection

We address three non-trivial challenges to leverage UCT for effective plan selection: (1) We model the problem of plan selection as an MDP (cf. Section 2). (2) We link the rewards in the MDP to the algorithm's runtime to achieve theoretical runtime guarantees (Section 5.3). (3) We propose methods to reuse partial results computed during a timeslice. This is required for both the runtime analysis and practical performance of Fast (Sections 5.3, 6.2, and 6.4).

Our MDP closely corresponds to the representation of the set of query plans $\mathcal{P}$ as a tree (cf. Figure 2): In the tree, nodes are distance functions or algorithms, edges to inner nodes are reductions, and root-leaf paths are query plans. In the MDP, nodes correspond to prefixes of such paths, actions are either reductions or algorithms, and transitions are the edges connecting the path prefixes. Rewards for all inner nodes are zero. The rewards on the leaf nodes represent the performance of the leaf's query plan.

5.3 Finding Efficient Query Plans

We first discuss a variant of Fast that assumes that the indexes required for all query plans $p \in \mathcal{P}$ are available. Conceptually, computing a (similarity) join requires comparing all pairs of records. Figure 3a visualizes all required comparisons for a similarity self-join. Due to the symmetry of distance functions, only the upper triangular matrix needs to be computed. To avoid the explicit comparison of all pairs of records, index-based similarity algorithms use an index that only returns promising pairs. Still, the area of the upper triangular matrix represents the full work required for the join.

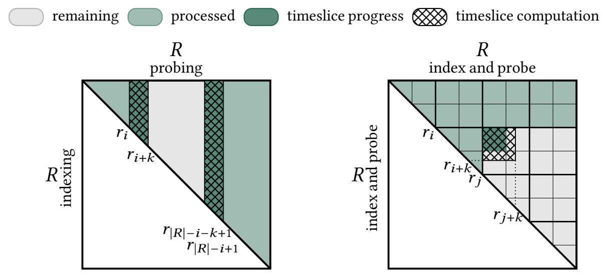
Next, we address the following problem of finding a good evaluation strategy: Given a set of query plans $\mathcal{P}$, e.g., computed by enumerating all paths starting from the query's distance node in the reduction graph (cf. Section 4), we want to evaluate the similarity join using some query plans $p \in \mathcal{P}$ that – in hindsight – approximately perform as well as the single optimal plan $p^* \in \mathcal{P}$. To this end, we use similar ideas as Trummer *et al.* [68] by evaluating partial joins with timeslices using different query plans. To compute a join, we first construct an UCT instance corresponding to the MDP of $\mathcal{P}$ (cf. Section 5.2). We then repeatedly select a plan $p \in \mathcal{P}$ using the UCT instance. A plan consists of (1) the reduction ρ implemented as the composition of any number of reduction functions, (2) a join operator implementing the filter step, and (3) a verification procedure. After selecting the current plan, the plan is executed by following the *reduce-filter-verify* framework for the next non-processed record r_i (cf. Figure 3a). We reduce r_i to $\rho(r_i)$ and probe $\rho(r_i)$ against the index on $\rho(R)$, obtaining a set of candidates. To avoid symmetric result pairs, we only probe $\rho(r_i)$ against the prefix of $\rho(R)$ consisting of the first $i-1$ records. According to the assumptions in Section 5.1, prefix index queries can be performed efficiently. To verify the results, we proceed as described in Section 4: Let $\rho = \rho_1 \circ \cdots \circ \rho_m$ and $\rho^k = \rho_1 \circ \cdots \circ \rho_k$ for $k \leq m$. We verify all candidates starting from the lowest level with reduction ρ^m and continue with the higher levels $\rho^{m-1}, \rho^{m-2}, \ldots$ until reaching the identity reduction ρ_0. The verification at level k removes all candidates s with $\rho^k(d)(\rho^k(r), \rho^k(s)) > \rho^k(\varepsilon)$.

Example 5.1. Consider a Tree Edit Distance (TED) lookup of r using the plan: TED❷SED❹Str.SS−Sym.SS−❖ For filtering, a tree is reduced to its traversal string, which itself is reduced to a set of q-grams with structural and then symmetric set similarity constraints ($\rho(r) = \rho_3(\rho_2(\rho_1(r)))$). A candidate s (retrieved by probing $\rho(r)$ against the prefix index) is then verified in stages: (1) Using symmetric set similarity $\rho^3(d)$ with threshold $\rho^3(\varepsilon)$ (corresponding to the full reduction $\rho_3 \circ \rho_2 \circ \rho_1$), (2) using structural set similarity $\rho^2(d)$ with threshold $\rho^2(\varepsilon)$, (3) using the string edit distance $\rho^1(d)$ with threshold $\rho^1(\varepsilon)$, and finally (4) TED $d(r_i, s)$ and ε (identity reduction ρ^0).

When probing from left to right (cf. Figure 3a), we probe against increasingly larger indexes that can yield larger candidate sets. To reduce this skew of increasing verification effort, we alternatingly pick a record from the left and the right side of the unprocessed records. We continue probing until we reach a timeout.

After the timeout, we evaluate the plan's performance and update the UCT instance. The chosen plan's performance for a timeslice is the number of conceptually compared pairs of records (cf. Figure 3a, hatched area). As we alternate between both sides of R, we have $m(|R|-1)$ non-reflexive and non-symmetric pairs after processing $2m$ records with m records from each side. As UCT expects rewards between 0 and 1, we normalize by the size of the upper triangular matrix $|R|(|R|-1)/2$. This results in the reward function $\frac{2m}{|R|}$.

Discussion. Prebuilding all possible indexes allows for a simple analysis. In particular, the query plans selected by UCT perform asymptotically as well as the optimal query plan. The key requirement for this result is that the work of all plans is shared: The progress of each plan towards computing the join result can be reused efficiently. Once a record has been probed by any query plan, it will not



(a) StatFast **(b) DynFast**

Figure 3: Processing pattern of StatFast and DynFast

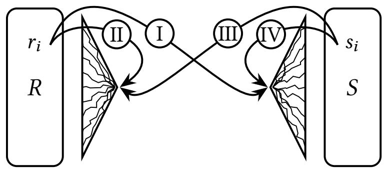


Figure 4: Two-sided incremental probe-and-insert

be probed again. This holds in the case of static indexes. For dynamic indexes as described in Section 6, some parts of the result must be discarded and recomputed later.

5.4 Regret Analysis

We analyze the expected regret of Fast with prebuilt indexes (called StatFast). Our goal is to perform approximately as well as the best query plan p^* in $\mathcal{P}$. In other words, we want to compare the policy of always selecting p^* for every timeslice to the plans selected by StatFast. This notion is exactly captured by *cumulative regret*, i.e., the difference in the sum over all rewards achieved during execution. As the reward for each timeslice is defined to be the size of the processed partial join, rewards measure the throughput of a policy. We will then relate throughput and execution time.

Regret Model and Assumptions. We denote optimal quantities with *-superscripts, e.g., our approach requires n timeslices while the optimal runtime is n^*. We analyze regret under the assumption that runtime is high compared to the number of plans. In particular, we assume that we achieve the asymptotic regret bounds of UCT and do not suffer from transitory regret [18]. Furthermore, we assume that the runtime related to traversing and updating the UCT tree and switching between plans is negligible. Finally, we assume that the average reward of the optimal algorithm cannot be arbitrarily small ($\Omega(1)$). To this end, timeslices are increased for large input sizes. This is in line with related work utilizing UCT in database systems [68].

Lemma 5.2. *StatFast has runtime regret of* $O(\log n)$.

We also express this additive regret in the more intuitive form of a multiplicative regret, bounding the asymptotic runtime ratio of Fast and the fastest algorithm.

Lemma 5.3. *StatFast has bounded expected multiplicative runtime regret* n/n^* *that converges to* 1 *with* $n \rightarrow \infty$.

6 DYNAMIC INDICES QUERY EVALUATION

In this section, we remove the assumption that the indexes of all query plans are built in advance. Our approach follows the typical procedure in similarity joins of alternatingly building and probing the index. After clarifying the assumptions, we introduce a dynamic version of Fast that builds indexes on-the-fly, analyze its regret, and reduce its overhead for practical applications.

6.1 Requirements on Similarity Join Algorithms

In addition to the requirements defined in Section 5.1, we require: (4) **Updatability:** Efficient incremental updates are supported: We can alternate between indexing new records and probing records without knowing all indexed records in advance. (5) **Range-Search:** In addition to index prefixes (cf. Assumption (3)), any contiguous range of record IDs can be probed efficiently. If an algorithm is used to index a dataset $R=\{r_i,...,r_l\}$, it supports efficient queries on any contiguous subset $R' = \{r_j, ..., r_k\}$ with $i \leq j \leq k \leq l$. Inverted list indexes support such queries by storing the lists sorted by record ID, finding the first record in the range using binary search, and skipping the remainder of the list after finding an ID higher than k.

6.2 Plan Evaluation with Index Construction

Missing prebuilt index structures poses an issue for evaluating the performance inside a timeslice. Index construction in itself does not constitute progress of the similarity join and thus should not be part of the reward. Otherwise, the index-construction part of the reward would be plan specific and would thus lead to regret bounds dependent on the number of query plans. Instead, we interleave building and probing the index to get a sample of the average performance of a specific query plan.

Incremental Probe-and-Index. When probing a record, the current state of the index will not encompass the full relation in general. Hence, we cannot simply partition the space of all record pairs like for the non-incremental setting in Figure 3a. Instead, we will alternate between building, probing, and verifying the next record. A simple implementation of this approach has been used in various similarity join algorithms [9, 19, 44, 79], but it can only be used to compute self-joins. We will refer to this approach as Inc1S (*incremental one-sided*) and introduce it as a special case of Inc2S (*incremental two-sided*) that we will discuss next. In our setting, Inc1S can only compute triangles along the diagonal in Figure 3b. For all other cases, we have to compute a non-self join between different parts of the relation using the following variation Inc2S that was already briefly discussed by Bouros *et al.* [13] in the context of the prefix filter. Figure 4 shows the overall structure. Given two relations R,S of equal size, we Ⅰ take the i-th record r_i from R, probe it against the current index on S, and verify the candidates. In step Ⅱ, we insert r_i into the index on R. Step Ⅲ and Ⅳ correspond to Ⅰ and Ⅱ with reversed roles of R and S. We proceed to the next record of both sides until both relations are processed. Inc1S works like Inc2S by setting $R=S$ and skipping steps Ⅲ and Ⅳ. Both Inc1S and Inc2S require the assumed updatability (cf. Section 6.1) of the index structure.

To conceptually compare all pairs of points in the upper triangular matrix in Figure 3b, we use both Inc1S and Inc2S. Given some pair of starting records (r_i,r_j), we either execute Inc1S if $i=j$ or Inc2S otherwise until a timeout is reached. After the timeout, we have

Algorithm 1: Block scheduling

```
Data: Stack $\mathcal{S}$, initially $((((1,1),(|R|,|R|)),\{UL,UR,LR\}))$
Function FullyProcessed():
    do pop from $\mathcal{S}$; remove popped subblock from $S^\top$
    while $S^\top$ fully processed
Function PartiallyProcessed(computed block $B_c$):
    while $S^\top$ not covered by $B_c$ do
        // blocks are processed from UL to LR
        push $UL$ subblock of $S^\top$ onto $\mathcal{S}$
    FullyProcessed()            // $S^\top$ was fully processed
Function NextBlock():
    if $S^\top$ partially processed then push its next subblock onto $\mathcal{S}$
    return $S^\top$
```

reached some record (r_{i+k},r_{j+k}) such that all pairs in the square with the corners (r_i,r_j) and (r_{i+k},r_{j+k}) have been evaluated (cf. Figure 3b). For self-joins using Inc1S, a triangle is computed. From now on, we will refer to both squares and triangles as *blocks*. Computing the full join requires *covering* the full upper triangular matrix with blocks. The sizes of the blocks will differ significantly both between different query plans and different parts of the matrix.

Block Scheduling. To avoid fragmentation and bookkeeping and guarantee covering all required pairs of records, we propose the following partitioning scheme of the matrix depicted in Algorithm 1. Algorithm 1 only encapsulates the scheduling of blocks; its functions are called as part of Fast. We maintain a stack $\mathcal{S}$ of blocks that still need to be filled. Each block consists of four subblocks UL,UR,LL,LR of equal size constructed by halving both sides of the block. In addition to the block, $\mathcal{S}$ also stores the set of subblocks of each block that have not been computed yet.

Initially, $\mathcal{S}$ contains the full dataset $((1,1),(|R|,|R|))$ with all subblocks $\{UL,UR,LR\}$; LL is missing to avoid symmetric pairs of self-joins. To schedule the next block (cf. Algorithm 1, `NextBlock`), we peek at the top $\mathcal{S}^\top$ of the stack. If $\mathcal{S}^\top$ has no computed subblocks, the full block $\mathcal{S}^\top$ is scheduled. Otherwise, the next subblock is pushed to $\mathcal{S}$ (becoming $\mathcal{S}^\top$) and scheduled.

After obtaining a block, it is processed using either Inc1S or Inc2S. There are two cases: (1) If the block is fully processed before reaching a timeout (cf. Algorithm 1, `FullyProcessed`), it is removed from $\mathcal{S}$ and the block's parents are updated. (2) Otherwise, the block was not fully processed before reaching a timeout (cf. Algorithm 1, `PartiallyProcessed`). This can only happen for the last block scheduled during a timeslice. In this case, the partially computed block is recursively split into its subblocks until the computed area of the partially computed block covers a subblock. Each subblock in the recursion is added to $\mathcal{S}$. Once these blocks reach the top of the stack again, their upper left subblock will have been computed already. Hence, we remove their UL subblock.

In the case of partially processed blocks, we discard the results from the partial block (cf. hatched area in Figure 3b) up to the last fully computed aligned subblock (resp. triangle, cf. shaded area in Figure 3b). We observe that the ratio of discarded results is bounded.

Lemma 6.1. *At most* 3/4 *of the pairs of tuples processed in any timeslice are discarded upon reaching a timeout.*

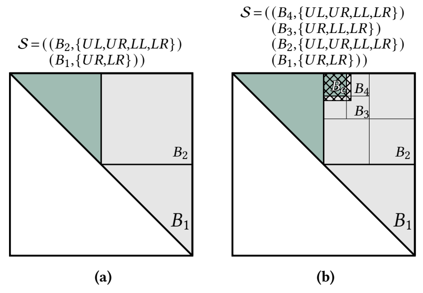


Figure 5: Example of block scheduling.

Example 6.2. Consider the block scheduling state depicted in Figure 5a. The UL subblock of B_1 was already fully processed. In the next step, B_2 is scheduled as it is the top of the stack $\mathcal{S}$. Figure 5b shows the progress B_c of computing B_2 after reaching a timeout, i.e., B_2 is only partially processed. As B_2 is only partially processed, B_2's UL subblock is partitioned two times until a resulting subblock is fully covered by B_c. The UL subblock of B_3 is fully processed, popped from $\mathcal{S}$, and B_4 is pushed to the stack to be scheduled as the next block.

Discussion. Discarding partially computed blocks simplifies the analysis and leads to at most constant multiplicative regret compared to the optimal algorithm choice by leveraging the bound on discarded result tuples. In practice, we continue the execution of Inc1S or Inc2S until the next larger subblock is fully processed. In these cases, the timeslice has varying size and the reward is the summed area over the measured time.

6.3 Regret Analysis

Assumptions. We analyze Fast with dynamic indexes (called DynFast) under the assumption that the throughput of each query plan approximates a stationary distribution. In particular, the throughput does not change when changing the size of the timeslice. To this end, we select larger timeslices for DynFast than for StatFast to reduce the skew introduced by disproportionately high indexing and probing cost compared to the verification cost for small blocks. We will discuss the size of the timeslice in Section 7.

Lemma 6.3. *DynFast has runtime regret of* $(1-1/4)n+O(\log n)$.

Lemma 6.4. *DynFast has bounded expected multiplicative runtime regret that converges to* 4 *with* $n\rightarrow\infty$.

6.4 Practical Considerations

Vectorization of Query Evaluation. In both StatFast and DynFast as described in Sections 5.3 and 6.2 , queries are evaluated record-by-record: A single record is reduced, probed, and candidates are verified. To reduce the overhead of performing multiple reductions, lookups inside an index, and one or more verification procedures, we partition all records into small *microbatches* and handle a microbatch of records in each invocation of a reduction, lookup, or verification. This style of *vectorized* evaluation of query plans is common in query engines [39].

Support for Index Updates. DynFast requires algorithms to support index updates in order to build indexes on-the-fly. In their original version, Prefix [9, 14], Palloc [19], PassJoin [44], and TJoin [29] assume some order on the records to build their index. All algorithms require their records to be sorted in ascending size and possibly lexicographically based on their content to break ties. In general, different algorithms might require incompatible orderings, making it impossible to sort the data only once for all algorithms. Additionally, if an algorithm requires a reduction, we initially do not know which parts of the dataset will be reduced and need to be ordered. To this end, we adapt all algorithms to support incremental updates in any order.

All algorithms Prefix, Palloc, PassJoin, and TJoin internally use a multi-level index structure. Some of these levels only perform point queries and some support range queries. As an example, a typical index implementing a prefix filter maps set elements to lists of record IDs sorted by record length. For a given set element (point query), the range of potentially similar records due to their size is scanned (range query). We replace each sorted list with an in-memory B^+ tree to support range queries and updates, while having high scan performance. All other algorithms are adapted similarly.

Caching of Indices. DynFast evaluates a plan on a block by building and probing the respective index, which is discarded afterwards. We propose a caching strategy to reduce the overhead of index construction. The cache keeps a set of indexes for each plan and can reuse an index if the plan is selected again for a different block.

We reuse an index structure under the following conditions given some to-be-computed block B: (1) If some index fully covers either side of B, we select the smallest among them. We select the smallest index to reduce the number of required skips while scanning the index. (2) If an index according to (1) does not exist, we search for an index that starts at the same offset into the dataset as B. Among all of them, we select the largest index. Selecting the largest index reduces the overhead of otherwise having to probe multiple indexes. (3) Otherwise, use either Inc1S or Inc2S to build a new index. Due to the assumption on efficient range search (cf. Assumption (5), Section 6.1), cases (1) and (2) can be computed efficiently.

While current standalone implementations of similarity joins typically build their index on-the-fly [9, 19, 29, 78], database systems usually prebuild index structures and maintain them under updates [58]. As a significant side effect of implementing index reuse, DynFast supports using prebuilt indexes to potentially speed up query evaluation. To this end, any existing index for some query plan is added to the cache before execution of DynFast. During computation, DynFast can choose between using existing indexes or building a new index structure for some other query plan from scratch depending on the performance of each query plan.

Caching of Intermediate Reduction Data and Signatures. For DynFast, the same microbatch of records might appear in multiple partial join computations for the same query plan. To compute the partial join, each record in the microbatch is reduced, probed and/or indexed, and verified. To reduce the cost of reducing the same microbatch multiple times, we cache the result of the reduction. As multiple query plans might share some of their reduction steps, the result of each reduction step is cached to speed up the evaluation of related query plans. As an example, consider the query plans for TED in Figure 2. Assume that the query plan TED❷SED❹Str.SS–Sym.SS–ⓒ is evaluated first. For the next timeslice, TED❷SED–ⓑ is selected. Due to their common prefix, the cached reduction data of SED is reused. In this case, no further reductions are necessary. Otherwise, only the remaining reductions along the path are performed.

Table 1: Dataset characteristics.

		#Documents	Document Size average	Document Size 99.9 pct.	Universe
		General datasets			
Sets	BMS-POS	$3.2\cdot10^5$	9.3	61	1657
Sets	Kosarak	$6.1\cdot10^5$	11.9	378	41270
Sets	DBLP14-Set	$5.2\cdot10^6$	77.7	209	24158
Sets	Lnonis1	$8.2\cdot10^6$	20.3	39	44103
Strings	DBLP-String	$1.4\cdot10^6$	106.3	294	37
Strings	Enron	$2.5\cdot10^5$	885.0	18332	37
Strings	Trec	$3.5\cdot10^5$	845.2	2793	37
Strings	Word	$1.2\cdot10^5$	9.7	19	26
Trees	DBLP-Tree	$3.9\cdot10^6$	26.1	81	$1.8\cdot10^7$
Trees	Python	$1.5\cdot10^5$	944.1	23682	$3.4\cdot10^6$
Trees	Sentiment	9645	37.3	99	19470
Trees	Swissprot	$3.2\cdot10^5$	998.7	9561	$7.4\cdot10^6$
		Supplementary datasets			
Strings	Gen20kl	$2.0\cdot10^4$	20000	20058	5
Strings	Gen50ks	$5.0\cdot10^4$	5000	5047	4
Strings	Trec_2	$2.3\cdot10^5$	1218.1	2834	37
Strings	Uniref	$4.0\cdot10^5$	446	3519	24
Trees	JScript1k	$3.9\cdot10^4$	8775.8	$3.3\cdot10^5$	$2.5\cdot10^6$
Trees	Python1k	$3.6\cdot10^4$	3012.6	$3.7\cdot10^4$	$2.6\cdot10^6$
Trees	Swissprot1k	$1.2\cdot10^5$	1902.0	$13\cdot10^4$	$6.4\cdot10^6$
Trees	SyntheticLU	$1.0\cdot10^6$	40	40	10000

To reduce the cost of probing the same microbatch against the same index multiple times, we further cache intermediate data required to probe the index. In similarity search, most algorithms (and in particular all algorithms implemented in Fast) are based on a *signature scheme* [57]. Intuitively, a signature scheme computes hash values for a given record. Depending on the algorithm, signature computation can be quite expensive. To reduce the burden of redundant signature computation for the same microbatch, we store its signatures in a cache. For our implementation, we use the *least recently used* replacement strategy for both caches.

7 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of Fast. We first describe the datasets and the implementation of Fast that we use in our evaluation. We then evaluate StatFast and DynFast, focusing on their ability to select efficient query plans in various settings. Afterwards, we show Fast's utility as a benchmark by comparing it against specialized state-of-the-art algorithms and demonstrate its extensibility.

7.1 Experimental Setup

We conducted all experiments on a cluster of servers, each having one AMD EPYC 9354P CPU with 32 cores and 384 GiB memory. We executed between 1-4 jobs on each server in parallel, depending on required memory. We implemented Fast, including all reductions and join operators, in C++. Our adaptations of the algorithms to support updates (cf. Section 6.4) have a negligible runtime impact for Palloc and PassJoin compared to preprocessing upfront. Dynamic updates for Prefix require dynamic token reassignments, leading to performance differences of less than ±15%, depending on the dataset and threshold, but the overall performance patterns remain the same. Missing data points correspond to timeouts (4 hours).

Datasets. We use twelve general and eight supplementary datasets in our evaluation. The general datasets are chosen from the pool of datasets used in previous works such that they cover diverse characteristics in document and universe size. The supplementary datasets are used to highlight specific aspects of an algorithm (e.g., due to skewed characteristics) and to reproduce experiments in Section 7.3. We refer to Mann *et al.* [47] for details on BMS-POS and Kosarak. DBLP14-Set[1] and Lnonis1 are described in Schmitt *et al.* [57]. The four general string datasets are from Jiang *et al.* [30]; the four general tree datasets are from Hütter *et al.* [29]. Gen20kl, Gen50ks, Trec$_2$, and Uniref are longer-string datasets from recent approximate string edit distance join works [33, 82, 83]. JScript1k, Python1k, and Swissprot1k [35] are subsets of larger tree datasets consisting only of large trees (≥1000 nodes). SyntheticLU is a synthetic tree dataset with a uniform and small (10^4 tokens) universe; all trees have height four and fanout three. Table 1 shows characteristics of all datasets.

Distance Functions. We evaluate FAST with Jaccard similarity on sets, the string edit distance on strings, and the tree edit distance on trees. We evaluated similar or larger ranges of distance thresholds ε compared to previous work. We study the extensibility of our approach to Jaro similarity in Section 7.4.

Algorithmic Variations. We compare both STATFAST (cf. Section 5) and DYNFAST (cf. Section 6) in a *warm* and a *cold* variation, i.e., four variations in total. In cold STATFAST, denoted STATFAST$_C$, all indexes have to be built at runtime; in warm STATFAST, denoted STATFAST$_W$, all indexes are prebuilt. For DYNFAST, DYNFAST$_C$ and DYNFAST$_W$ refer to DYNFAST with empty and filled (index, reduction, signature) caches, respectively. The caches are warmed up by executing the same join twice and only measuring the second execution.

Algorithmic Parameters. We set all cache sizes (index, reduction, intermediate data) to $2|R|$. We exclude Euler strings in Figure 1 due to their similarity to traversal strings. We exclude TJOIN for dynamic experiments due to its high similarity to label sets with PREFIX. We use TJOIN's verification procedure for all tree plans. We set $q = 3$ for the q-gram reduction. We set the timeslice to 0.2 seconds for STATFAST. In DYNFAST, we set timeslices for INC1S and INC2S differently, using a timeslice of 0.2 and 0.8, respectively, to reduce skew as stated in Section 6. Inspired by the scaling experiments in Trummer *et al.* [68], we set the exploration weight w (cf. Section 2) proportionally to the achieved rewards. While Trummer *et al.* [68] use a fixed weight, we dynamically update w at runtime. UCT requires random variables in $[0,1]$ and is often used with a weight of $\sqrt{2}$ [68]. In FAST, rewards represent relative throughput and tend to range between 0 and 10^{-4}. For such small rewards, high exploration weights lead to long convergence times. To this end, we scale the exploration weight depending on the observed rewards, initially setting $w = 0.01\sqrt{2}$ and updating to $w = \sqrt{2}r_{max}$ with exponentially increasing intervals between each update for the highest observed reward r_{max}.

7.2 Evaluating FAST's Standalone Performance

We first evaluate FAST's internals, focusing on its ability to select query plans using static (STATFAST) and dynamic (DYNFAST) indexes.

7.2.1 Evaluation of STATFAST. We evaluate the ability of STATFAST to select highly performant query plans. To this end, we compare STATFAST against a standalone execution of each individual query plan. In this setting, we assume that all indexes are prebuilt and only measure the time to compute the join. Figure 6 reports the results on all general datasets. A query plan is named after its reduction step(s) (Q: q-grams, T: traversal strings, L: label sets) and the join algorithm.

In general, the best query plan *differs* for different datasets and thresholds. On set data, PALLOC performs best in most settings, only being outperformed by PREFIX on high thresholds in Kosarak.

On strings, PASSJOIN and Q-PREFIX compete for the best query plan. On datasets with short strings (DBLP-String and Word), PASSJOIN is the most selective query plan and requires few verifications. For longer strings and higher thresholds (Enron and Trec), the lightweight Q-PREFIX approach is selective enough.

For trees, the label-set-based query plans and in particular L-PREFIX are the fastest on all datasets except Sentiment due to their low overhead. This is due to the large universe size; many uncommon tokens are readily leveraged by PREFIX and PALLOC to filter candidates. Sentiment has a smaller universe size than other datasets, requiring methods that preserve more of the tree structure than label sets. Hence, traversal-string-based approaches (in particular T-PASSJOIN) slightly outperform the other plans. Motivated by this observation, we additionally evaluate STATFAST on the supplementary dataset SyntheticLU (not shown) to compare the performance of query plans on a dataset with a small universe size. On that dataset, plans based on the prefix filter (PREFIX and TJoin) are outperformed by the more selective T-PASSJOIN and L-PALLOC by up to three orders of magnitude.

STATFAST closely follows the best performing query plan. On average over all thresholds and datasets, STATFAST is only 12% slower than the fastest query plan. The worst case happens for extremely short runtimes on the Sentiment dataset at threshold $\varepsilon = 2$, where STATFAST requires 0.03 seconds compared to the optimal 0.012 seconds (factor 2.5). When only considering runtimes of at least 1 second, the average and maximum slowdown of STATFAST shrink to 7% and 38%, respectively. To summarize, STATFAST effectively selects the fastest query plan and has little overhead.

7.2.2 Analysis of Plan Selection. On its path to convergence, UCT will inevitably also select non-optimal plans for a few timeslices. However, the majority of timeslices - over 90% in most settings - are executed using a single, optimal query plan. In cases where overall execution time is very short and multiple algorithms have comparable performance, no consensus is reached before termination. This behavior is observed for high ϵ values on BMS-POS and Kosarak, and for low ϵ values on Sentiment and Python.

With the exception of TJOIN (due to its similarity to L-PREFIX), all algorithms and reductions are required for the most efficient query plan in at least one setting: PREFIX: Kosarak (high ε), Enron, Trec, DBLP-Tree, Swissprot; PALLOC: BMS-POS, Kosarak (low ε), DBLP14-Set, Lnonis1, Sentiment (low ε); PASSJOIN: DBLP-String, Word, Sentiment (high ε), SyntheticLU; *Q-grams*: Enron, Trec; *Traversal strings*: Sentiment (high ε); *Label sets*: DBLP-Tree, Swissprot, Sentiment (low ε).

7.2.3 Evaluation of Algorithmic Variations. We now evaluate the cold and warm variants of DYNFAST and STATFAST (cf. Section 7.1).

Runtime Evaluation. Figure 7 shows the runtime performance of all four variations. We expect the following descending order of performance for the different scenarios: (1) STATFAST$_W$ does not require

[1]DBLP14-Set uses the current DBLP version [65] with preprocessing from [57, 76].

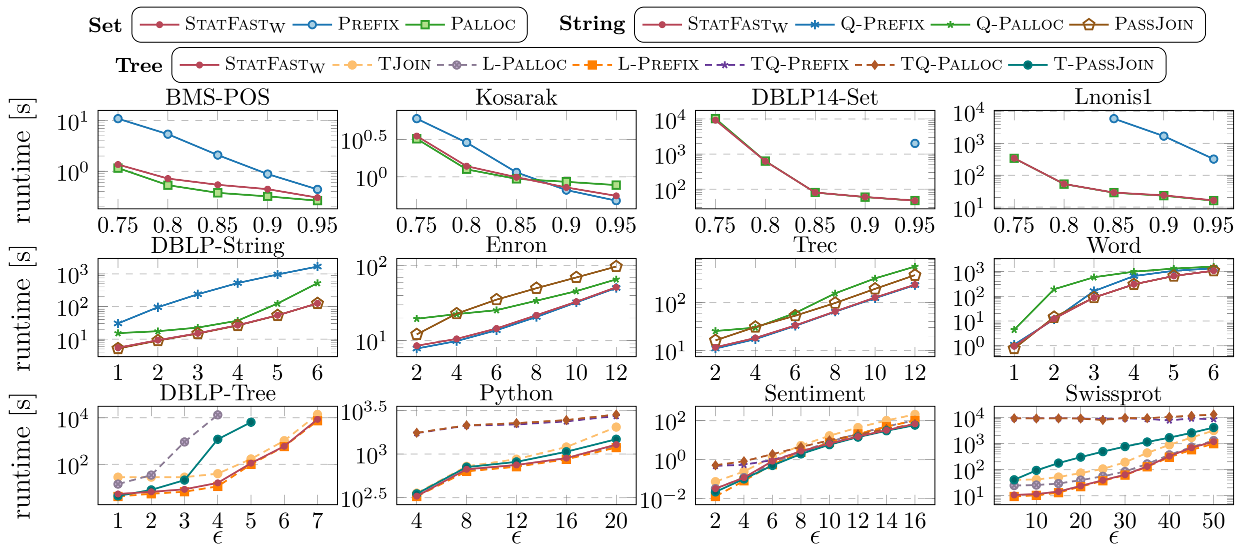


Figure 6: Comparison of StatFast against all individual query plans.

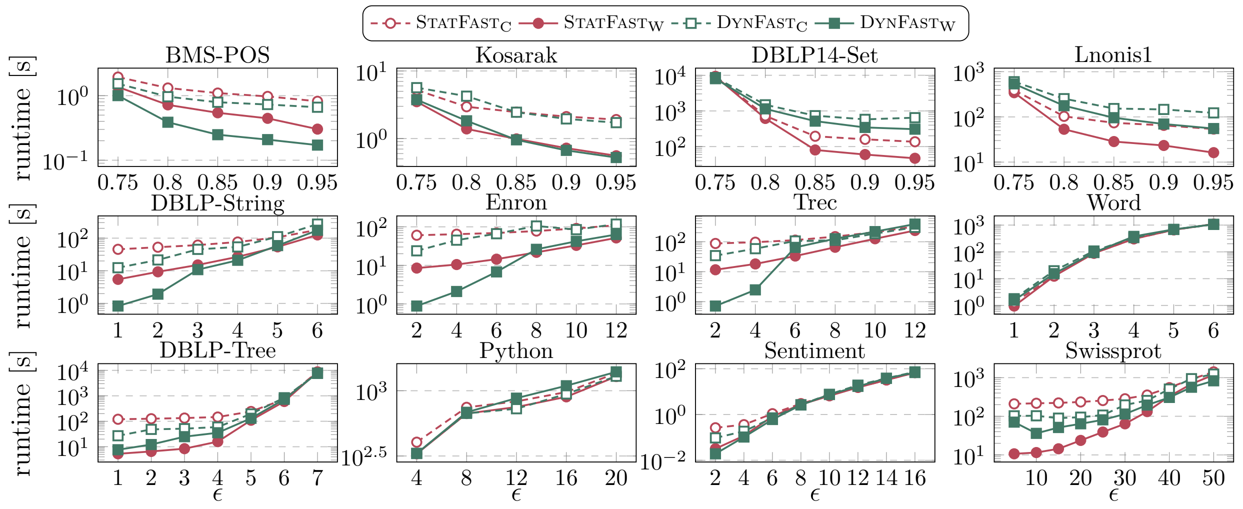


Figure 7: Comparison of StatFast and DynFast.

Table 2: Index redundancy of StatFast vs. DynFast.

Dataset	redund.	% red.	Dataset	redund.	% red.
BMS-POS	0	−100%	Trec	0.32	−84%
Kosarak	0.60	−40%	Word	1.04	−48%
DBLP14-Set	0.14	−86%	DBLP-Tree	0.47	−91%
Lnonis1	0.08	−92%	Python	1.7	−66%
DBLP-String	0.16	−92%	Sentiment	0.95	−81%
Enron	0.30	−85%	Swissprot	0.55	−89%

indexing and has less overhead than DynFast. (2) DynFast$_W$ can leverage its caches to avoid most index construction work and reduce the overhead of DynFast. (3) DynFast$_C$ starts with an empty index cache, but does not construct all indexes. (4) StatFast$_C$ constructs all indexes and is unsuitable if the number of query plans is high.

The results in Figure 7 are consistent with our expectation for most datasets (Kosarak, Word, DBLP-Tree, Python, Sentiment, Swissprot, high ε for DBLP-String, Enron, and Trec). For low distance thresholds ε, DynFast$_W$ outperforms StatFast$_W$ on BMS-POS (similarity $\varepsilon \geq 0.75$), DBLP-String ($\varepsilon \leq 2$), Enron ($\varepsilon \leq 6$), and Trec ($\varepsilon \leq 4$). This result is an effect of caching at the hardware level: For low thresholds, indexes are small and might fit into the processor's cache. In DynFast$_W$, we execute the same query twice and thus access the same index structures that are already in the processor cache. When flushing the hardware cache in between runs, the difference between DynFast$_W$ and StatFast$_W$ decreases for all results affected by this phenomenon. On DBLP14-Set and Lnonis1, for high thresholds, probing is expensive compared to verification. In general, DynFast requires probing the same record multiple times against small, partial indexes. Even with caching of intermediate reduction data and signatures, DynFast falls behind StatFast on these datasets. On average over all executions, DynFast$_W$, DynFast$_C$, and StatFast$_C$ are 58%, 197%, and 306% slower than StatFast$_W$.

Evaluation of Index Redundancy. We further analyze the *index redundancy* of DynFast compared to StatFast. We define the index redundancy as the average number of *additional indexes* that a single record is indexed in. For StatFast, each query plan corresponds to an index and thus its index redundancy is $|\mathcal{P}|-1$. DynFast avoids building major parts of inefficient indexes, only building highly performant ones. Table 2 lists the average index redundancy of DynFast and the relative reduction to StatFast on all datasets. On average, DynFast requires 80%, 77%, and 82% fewer additional indexes on sets, strings, and trees, respectively.

In summary, our results show that (1) DynFast$_W$ has lower index redundancy at the cost of a moderate runtime penalty compared to StatFast$_W$ due to its probing overhead, and (2) DynFast$_C$ outperforms StatFast$_C$ both in terms of runtime and index redundancy.

7.3 Fast as a Baseline

We now evaluate the performance of Fast as a baseline for new join algorithms. We therefore consider three recently proposed algorithms for sets, strings, and trees and compare them to Fast. To this end, we compare against StatFast$_C$ (slowest variation on average, cf. Section 7.2.3) for algorithms that we outperform and StatFast$_W$ (fastest variation on average, cf. Section 7.2.3) for algorithms that outperform us. We omit the remaining variations of Fast to avoid clutter.

7.3.1 Comparison against TwoL (Sets). TwoL [57] was recently proposed as a similarity join algorithm on sets for symmetric set similarity functions. Its main contribution is its ability to combine two index structures for set similarity and adapt to the dataset's characteristics. Compared to Fast, it only supports similarity search on sets and is based on optimizing a cost model estimating the cost of a join using either parts of individual indexes or both at the same time. It promises high performance on datasets with mixed characteristics.

Experimental Setup and Results. We compare the author's implementation of TwoL against StatFast$_C$ on all set datasets (cf. Table 1). Compared to Fast, TwoL requires preprocessed datasets. The preprocessing time is not included in our results. We compare on Jaccard similarity. For Jaccard, TwoL combines Prefix with Palloc. Similarly, Fast can select the Prefix or the Palloc plan.

Figure 8 shows the runtime (left axis) of TwoL and StatFast$_C$; Prefix and Palloc are included for reference. TwoL and Fast perform nearly identically. They both effectively avoid the highly inefficient choice of Prefix on DBLP14-Set and Lnonis1. The right axis of Figure 8 shows the usage ratio of Palloc (#timeslices for Fast, #reindexed records for TwoL) in Fast and TwoL. Both algorithms are clearly correlated in their choices.

Conclusion. Fast behaves nearly identically to TwoL both in terms of runtime and index decisions. Hence, including TwoL into Fast will not yield any significant improvement.

7.3.2 Comparison against MinJoin++ (Strings). MinJoin++ [33] is an approximate algorithm for the string edit distance. It is based on splitting all strings into disjoint substrings using pivot points selected with random hash functions on q-grams. Compared to other algorithms like PassJoin, the number of substrings is linear in ε. MinJoin++ focuses on long strings and promises finding all pairs with high probability.

Experimental Setup and Results. We compare MinJoin++ against Fast on all four general string datasets (cf. Table 1) and four supplementary string datasets (Gen20kl, Gen50ks, Trec$_2$, Uniref; cf. Table 1) that were used in the original evaluation of MinJoin++ [33]. We use the author's implementation of MinJoin++ [34] and set all parameters according to the author's default suggestions [33]. In addition to 3-grams included in Fast by default (cf. Section 7), we add additional q-gram reductions for $q \in \{5,7,9,11,13,15\}$ due to the small universe sizes of the genetic datasets (Gen20kl, Gen50ks).

Figure 9 shows the runtime of MinJoin++ and StatFast$_W$ on DBLP-String, Word, Gen20kl, and Uniref. The performance on Enron and Trec (not shown) is similar to Word, while the performance on Gen50ks and Trec$_2$ (not shown) is similar to Gen20kl and Uniref, respectively. For MinJoin++, we additionally show the recall for each data point. On datasets with very long strings and high thresholds (Gen20kl, Gen50ks, Trec$_2$, Uniref), MinJoin++ significantly outperforms Fast while achieving high recall (>99%). For datasets with short strings (Word, DBLP-String) and for larger thresholds on Enron and Trec, MinJoin++ only achieves low recalls (<50%) in many cases. This is a known limitation of randomization used in MinJoin++ [33].

Conclusion. The performance of MinJoin++ heavily depends on the dataset. On datasets with short strings, MinJoin++ achieves low recall and high runtimes; on datasets with long strings, MinJoin++ finds all result pairs with a significant speedup compared to Fast. Therefore, we consider MinJoin++ a worthwhile candidate for inclusion in the reduction graph of Fast.

7.3.3 Comparison against SyncSig (Trees). SyncSig [35] introduces two approximate algorithms (BJoin and EJoin) for TED. Both are similar to MinJoin and either partition the trees into balls (BJoin) or partition the Euler strings of the trees (EJoin). Similar to MinJoin, both algorithms claim high performance for large trees.

Experimental Setup and Results. SyncSig was already compared to TJoin on the general datasets (cf. Table 1) in the original work [35]. For these datasets, all algorithms performed similarly: They all must verify at least the true-positive trees and the verification step is responsible for most of the runtime. The authors thus remove all trees smaller than 1000 nodes to reduce the output size. We use their datasets (JScript1k, Python1k, Swissprot1k; cf. Table 1) for this evaluation. We use the author's implementation of SyncSig [36].

Figure 10 shows the runtime of SyncSig in both its BJoin and EJoin variant and StatFast (from scratch, i.e., including indexing time). After the execution of StatFast, we observe that the query plan TED-❶-Str.SS–Sym.SS–◆d[2] outperforms all other query plans significantly. We then measure only the performance of this single query plan (referred to as *StatFast (partition only)* in Figure 10). StatFast (partition only) always outperforms BJoin and shows similar performance to EJoin. The remaining difference in runtime on JScript1k and Python1k can be attributed to the materialization of data reductions: By computing Palloc directly on the nodes of the tree without materializing them as sets, this gap can be reduced.

Conclusion. To the best of our knowledge, no existing algorithm for TED uses a query plan similar to the Palloc-based plan of Fast.

[2]The variation of Palloc used here disables deletion neighborhoods and is equivalent to the partition filter in PartEnum [2].

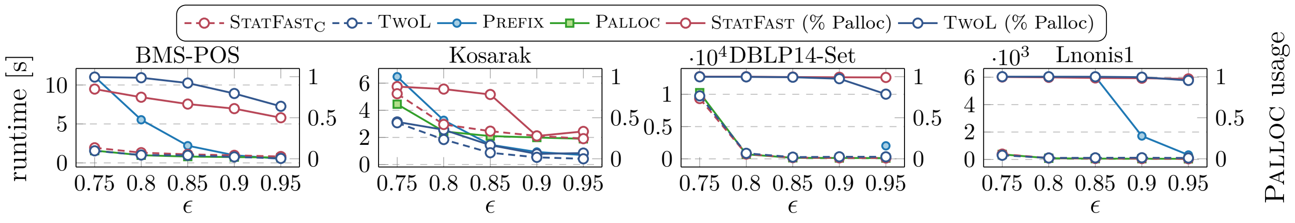


Figure 8: Runtime comparison of Fast against TwoL.

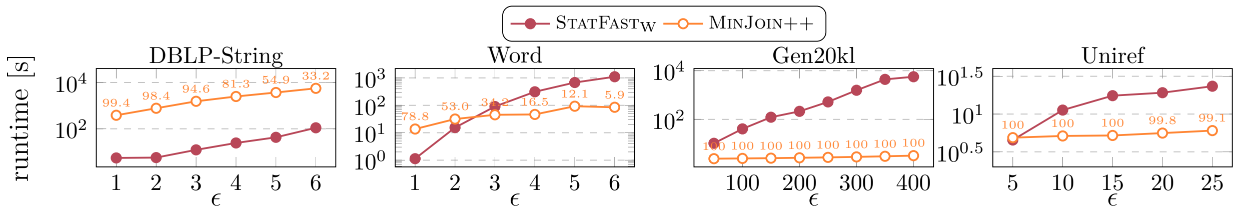


Figure 9: Runtime comparison of Fast against MinJoin++.

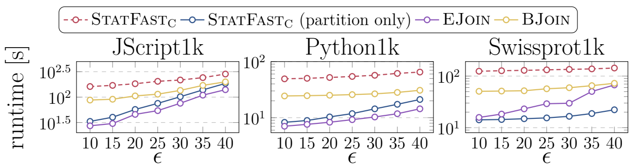


Figure 10: Runtime comparison of Fast against SyncSig.

This query plan is efficient and performs approximately as well as the better variation of SyncSig on all tested datasets. Hence, the addition of SyncSig to Fast would not yield a significant improvement.

7.4 Fast as a Tool for Prototyping

We show the extensibility of Fast to an unsupported similarity function as a method to design a similarity join algorithm. To this end, we demonstrate the inclusion of the Jaro similarity on strings and show the required steps to implement baseline support for Jaro inside Fast using reductions. Furthermore, we compare Fast against Limes [22, 50], a state-of-the-art algorithm for Jaro similarity joins.

Jaro Similarity. The Jaro similarity is a measure defined on strings. For two strings r,s, we define the number of matches m as the number of identical characters that appear at most $\frac{\max\{|r|,|s|\}}{2}$ character positions apart from each other in r and s. Furthermore, we define the number of transpositions t as the cardinality of the subset of matches that are not matched at the same position in both strings. Then, the Jaro similarity is defined as $d_j(r,s) = \frac{1}{3}(\frac{m}{|r|} + \frac{m}{|s|} + \frac{m-t}{m})$. Jaro similarity is used in entity linking and related fields [17, 22, 52].

Implementing Jaro using Reductions. Besides an implementation of the similarity function itself, a reduction to a known distance function with known query plans is required. To this end, we use a common bound of d_j based on the overlap of the character sets of two strings [22, 74]: Define *Jaro Overlap* similarity as $d_{jo} = \frac{1}{3}(\frac{|r\cap s|}{|r|} + \frac{|r\cap s|}{|s|} + 1)$, where the strings r, s are interpreted as sets of characters. Then $d_j(r,s) \leq d_{jo}(r,s)$. Jaro Overlap is an instance of a symmetric set similarity, with the equivalent overlap defined as $eqo_{jo}(r,s,\varepsilon) = (3\varepsilon - 1)\frac{|r|\cdot|s|}{|r|+|s|}$. Thus, Jaro Overlap does not require any significant implementation effort as it is defined by its equivalent overlap like all other symmetric set similarities.

To summarize, implementing Jaro similarity inside Fast only requires (1) implementing Jaro similarity for pairs of strings, (2) adding a new equivalent overlap function to support Jaro Overlap, (3) implementing the *character sets* reduction, mapping strings to sets (implemented as a special case of q-grams with $q = 1$), and (4) registering Jaro, Jaro Overlap, and character sets in the reduction graph (cf. Figure 1). Our reduction graph finds two query plans for Jaro similarity: Jaro is reduced to Jaro Overlap using character sets; then, Jaro Overlap is evaluated using Prefix or Palloc.

Experimental Evaluation. We compare Fast against Limes [22, 50], a state-of-the-art algorithm for Jaro similarity joins. We adapt the original implementation of Limes [23]: (1) We reimplement the original Java implementation of Limes in C++, (2) adjusted an incorrect early termination condition that was previously noticed by Keil *et al.* [37], and (3) removed the use of redundant data structures. Except for modification (2) that is required for correctness, all modifications increased the performance of Limes.

We evaluate Fast and Limes on all general datasets. Due to high runtimes for low Jaro thresholds on these datasets, we take samples such that the runtime of Limes does not significantly exceed one hour. For DBLP-String, Enron, and Trec, the samples are of size 60000, 11000, and 8000, respectively. We use the full Word dataset.

Figure 11 shows the runtime of Limes and StatFast$_C$. Except for low thresholds on Enron and Trec, StatFast significantly outperforms Limes for all datasets. Limes uses early termination and grouped verification, but essentially compares all pairs of strings. Fast leverages both Prefix and Palloc to effectively prune dissimilar pairs and thus does not need to consider all pairs of strings. In particular, Fast tends to use Prefix for filtering of low thresholds (up to 0.88–0.94) and Palloc for high thresholds (0.88–0.94 and higher).

To summarize, extending Fast to Jaro similarity only requires adding the Jaro similarity function, an equivalent overlap function,

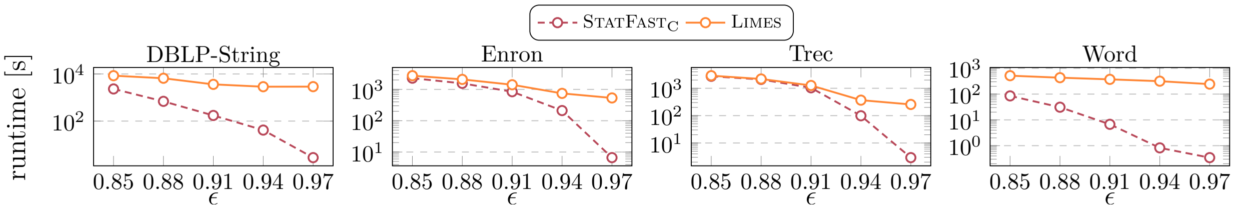


Figure 11: Runtime comparison of Fast against Limes.

a reduction, and new entries in the reduction graph. In particular, no new algorithms are required. This prototype serves as a baseline for algorithms using character sets to bound Jaro similarity. Fast outperforms the current state-of-the-art algorithm for Jaro joins Limes by one to three orders of magnitude in most settings.

7.5 Main Experimental Insights

Our experiments (Sections 7.2-7.4) highlight various strengths of Fast. First, Fast automatically selects near-optimal query plans with low overhead (on avg. only 12% slower than optimal plans, cf. Section 7.2.1) and utilizes various algorithmic primitives based on dataset characteristics (cf. Section 7.2.2). Second, Fast offers flexible deployment via StatFast and DynFast variations, supporting systems with pre-built (StatFast$_W$) indexes for frequently queried data or on-the-fly construction with caching (DynFast) for infrequently queried data (cf. Section 7.2.3). Third, Fast serves as a robust baseline, matching or exceeding the performance of specialized algorithms using query plans not yet reported in the literature (e.g., a novel Palloc-based plan for TED, cf. Section 7.3.3). Finally, its modular design facilitates rapid prototyping and extension, as shown by the Jaro similarity case study where Fast significantly outperformed a state-of-the-art competitor with minimal coding efforts (cf. Section 7.4). Collectively, these findings show that Fast is a robust and extensible system for similarity query evaluation.

8 RELATED WORK

Standalone Algorithms. A large number of standalone algorithms for similarity joins has been proposed in the literature. These works are orthogonal to our work as, to the best of our knowledge, all current algorithms can be included into Fast to extend it with new query plans.

Most algorithms for set similarity use the prefix filter [14] or the partition-and-enumeration framework [2]. We refer to the survey by Mann *et al.* [47] for details on the numerous variations of prefix-based algorithms [9, 13, 14, 46, 70, 79]. Further developments include grouping of index entries and probing signatures [72], efficiently enumerating subsets as signatures [20], and techniques to combine multiple algorithms depending on the dataset [57, 76].

For string similarity joins, most algorithms focus on the string edit distance and we refer to Jiang *et al.* [30] for details. Recent developments for approximate string similarity joins include embedding SED in Hamming space [82] and splitting the strings at pivot points determined by a randomized function [33, 83].

In the context of tree edit distance joins, Li *et al.* [43] survey bounds based on histograms [32], strings [27], and subtrees [80]. Further developments include splitting the trees into subgraphs and applying the pigeonhole principle [66], using enhanced label sets including positional information [29], and approximate approaches based on tree partitioning or Euler string partitioning [35].

Similarity Joins in Systems. Recently, some systems taking a more holistic approach to similarity joins have been proposed. SIREN [8] leverages a traditional DBMS and handles similarity queries externally. DIMA [62, 63] implements Palloc [19] in Apache Spark and extends it for distributed computation. DIMA performs cost-based optimization either using a prebuilt index or by sampling. Silva *et al.* [59] study different similarity operators, their conceptual evaluation, and rewrite rules for query optimization. Wang *et al.* [75] use machine learning for cardinality estimation of similarity joins. Sun *et al.* [61] reduce the dimensionality of their training data by partitioning and learning local models. Unlike previous work, we focus on an extensible system for different similarity functions and domains. Note that selecting an efficient query plan is orthogonal to estimating the cardinality of the result of the similarity join. While all joins have the same cardinality, different similarity join algorithms greatly vary in cost.

Feedback-based Query Evaluation. Several query plan selection schemes using measured runtime performance have been proposed recently. Eddies [7] routes input tuples to the plan's operators based on lottery scheduling [69] without formal guarantees. Li *et al.* [45] dynamically change join orders based on observed selectivities. Trummer *et al.* [68] use UCT [41] to find efficient join orders during execution. Compared to Trummer, our work on similarity queries introduces a novel technique for query plan enumeration, requires different MDP modeling, and different strategies for work sharing.

9 CONCLUSION

We introduced Fast, a similarity join algorithm that is both extensible and efficient, overcoming limitations of current monolithic solutions. By leveraging reductions to transform complex join problems into simpler ones, our system efficiently enumerates and selects optimal query plans via a sampling-based strategy without the need for cost models. Supporting both prebuilt and on-the-fly indexing with effective caching, Fast achieves performance near the best individual plan in our extensive evaluation. Finally, our case studies further demonstrate Fast's extensibility and potential as a baseline.

ACKNOWLEDGMENTS

This research was funded in whole, or in part, by the Austrian Science Fund (FWF) 10.55776/P34962, the State of Salzburg, Austria: Culture and Science Department, reference number 20204-WISS/262/9-2021, and the Austrian ministries BMIMI, BMWET, and the State of Upper Austria in the frame of the SCCH COMET competence center INTEGRATE (FFG 892418).

REFERENCES

[1] Tatsuya Akutsu. 2006. A relation between edit distance for ordered trees and edit distance for Euler strings. *Inf. Process. Lett.* 100, 3 (2006), 105–109. https://doi.org/10.1016/j.ipl.2006.06.002

[2] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 918–929. http://dl.acm.org/citation.cfm?id=1164206

[3] Nikolaus Augsten. 2018. A Roadmap towards Declarative Similarity Queries. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose (Eds.). OpenProceedings.org, 509–512. https://doi.org/10.5441/002/EDBT.2018.59

[4] Nikolaus Augsten and Michael H. Böhlen. 2013. *Similarity Joins in Relational Database Systems*. Morgan & Claypool Publishers. https://doi.org/10.2200/S00544ED1V01Y201310DTM038

[5] Nikolaus Augsten, Michael H. Böhlen, and Johann Gamper. 2005. Approximate Matching of Hierarchical Data Using pq-Grams. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, 301–312. http://www.vldb.org/archives/website/2005/program/paper/wed/p301-augsten.pdf

[6] Nikolaus Augsten, Michael H. Böhlen, and Johann Gamper. 2010. The *pq*-gram distance between ordered labeled trees. *ACM Trans. Database Syst.* 35, 1 (2010), 4:1–4:36. https://doi.org/10.1145/1670243.1670247

[7] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein (Eds.). ACM, 261–272. https://doi.org/10.1145/342009.335420

[8] Maria Camila Nardini Barioni, Humberto Luiz Razente, Agma J. M. Traina, and Caetano Traina Jr. 2006. SIREN: A Similarity Retrieval Engine for Complex Data. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 1155–1158. http://dl.acm.org/citation.cfm?id=1164232

[9] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy (Eds.). ACM, 131–140. https://doi.org/10.1145/1242572.1242591

[10] Thomas Bocek, Ela Hunt, David Hausheer, and Burkhard Stiller. 2008. Fast similarity search in peer-to-peer networks. In *IEEE/IFIP Network Operations and Management Symposium: Pervasive Management for Ubioquitous Networks and Services, NOMS 2008, 7-11 April 2008, Salvador, Bahia, Brazil*, Marcus Brunner, Carlos Becker Westphall, and Lisandro Zambenedetti Granville (Eds.). IEEE, 240–247. https://doi.org/10.1109/NOMS.2008.4575140

[11] Christian Böhm, Bernhard Braunmüller, Markus M. Breunig, and Hans-Peter Kriegel. 2000. High Performance Clustering Based on the Similarity Join. In *Proceedings of the 2000 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 6-11, 2000.* ACM, 298–305. https://doi.org/10.1145/354756.354832

[12] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. 2001. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, Sharad Mehrotra and Timos K. Sellis (Eds.). ACM, 379–388. https://doi.org/10.1145/375663.375714

[13] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. 2012. Spatio-textual similarity joins. *Proc. VLDB Endow.* 6, 1 (2012), 1–12. https://doi.org/10.14778/2428536.2428537

[14] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang (Eds.). IEEE Computer Society, 5. https://doi.org/10.1109/ICDE.2006.9

[15] Tobias Christiani and Rasmus Pagh. 2017. Set similarity search beyond MinHash. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, Hamed Hatami, Pierre McKenzie, and Valerie King (Eds.). ACM, 1094–1107. https://doi.org/10.1145/3055399.3055443

[16] William W. Cohen. 2000. Data integration using similarity joins and a word-based information representation language. *ACM Trans. Inf. Syst.* 18, 3 (2000), 288–321. https://doi.org/10.1145/352595.352598

[17] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. 2003. A Comparison of String Distance Metrics for Name-Matching Tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), August 9-10, 2003, Acapulco, Mexico*, Subbarao Kambhampati and Craig A. Knoblock (Eds.). 73–78. http://www.isi.edu/info-agents/workshops/ijcai03/papers/Cohen-p.pdf

[18] Pierre-Arnaud Coquelin and Rémi Munos. 2007. Bandit Algorithms for Tree Search. In *UAI 2007, Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence, Vancouver, BC, Canada, July 19-22, 2007*, Ronald Parr and Linda C. van der Gaag (Eds.). AUAI Press, 67–74. https://doi.org/10.5555/3020488.3020497

[19] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. 2015. An Efficient Partition Based Method for Exact Set Similarity Joins. *Proc. VLDB Endow.* 9, 4 (2015), 360–371. https://doi.org/10.14778/2856318.2856330

[20] Dong Deng, Yufei Tao, and Guoliang Li. 2018. Overlap Set Similarity Joins with Theoretical Guarantees. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 905–920. https://doi.org/10.1145/3183713.3183748

[21] Carmel Domshlak and Zohar Feldman. 2013. To UCT, or not to UCT? (Position Paper). In *Proceedings of the Sixth Annual Symposium on Combinatorial Search, SOCS 2013, Leavenworth, Washington, USA, July 11-13, 2013*, Malte Helmert and Gabriele Röger (Eds.). AAAI Press, 63–70. https://doi.org/10.1609/SOCS.V4I1.18299

[22] Kevin Dreßler and Axel-Cyrille Ngonga Ngomo. 2017. On the efficient execution of bounded Jaro-Winkler distances. *Semantic Web* 8, 2 (2017), 185–196. https://doi.org/10.3233/SW-150209

[23] Kevin Dreßler and Axel-Cyrille Ngonga Ngomo. 2017. On the efficient execution of bounded Jaro-Winkler distances (Source Code). https://github.com/dice-group/LIMES. Source code accompanying the paper.

[24] Zohar Feldman and Carmel Domshlak. 2013. Monte-Carlo Planning: Theoretically Fast Convergence Meets Practical Efficiency. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence, UAI 2013, Bellevue, WA, USA, August 11-15, 2013*, Ann E. Nicholson and Padhraic Smyth (Eds.). AUAI Press. https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=2382&proceeding_id=29

[25] Zohar Feldman and Carmel Domshlak. 2014. Simple Regret Optimization in Online Planning for Markov Decision Processes. *J. Artif. Intell. Res.* 51 (2014), 165–205. https://doi.org/10.1613/JAIR.4432

[26] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate String Joins in a Database (Almost) for Free. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann, 491–500. http://www.vldb.org/conf/2001/P491.pdf

[27] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. 2002. Approximate XML joins. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki (Eds.). ACM, 287–298. https://doi.org/10.1145/564691.564725

[28] Thomas Hütter, Nikolaus Augsten, Christoph M. Kirsch, Michael J. Carey, and Chen Li. 2022. JEDI: These aren't the JSON documents you're looking for?. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1584–1597. https://doi.org/10.1145/3514221.3517850

[29] Thomas Hütter, Mateusz Pawlik, Robert Loschinger, and Nikolaus Augsten. 2019. Effective Filters and Linear Time Verification for Tree Similarity Joins. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 854–865. https://doi.org/10.1109/ICDE.2019.00081

[30] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String Similarity Joins: An Experimental Evaluation. *Proc. VLDB Endow.* 7, 8 (2014), 625–636. https://doi.org/10.14778/2732296.2732299

[31] Liang Jin, Chen Li, and Rares Vernica. 2008. SEPIA: estimating selectivities of approximate string predicates in large Databases. *VLDB J.* 17, 5 (2008), 1213–1229. https://doi.org/10.1007/S00778-007-0061-2

[32] Karin Kailing, Hans-Peter Kriegel, Stefan Schönauer, and Thomas Seidl. 2004. Efficient Similarity Search for Hierarchical Data in Large Databases. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings (Lecture Notes in Computer Science)*, Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari (Eds.), Vol. 2992. Springer, 676–693. https://doi.org/10.1007/978-3-540-24741-8_39

[33] Nikolai Karpov, Haoyu Zhang, and Qin Zhang. 2024. MinJoin++: a fast algorithm for string similarity joins under edit distance. *VLDB J.* 33, 2 (2024), 281–299. https://doi.org/10.1007/S00778-023-00806-Z

[34] Nikolai Karpov, Haoyu Zhang, and Qin Zhang. 2024. MinJoin++: a fast algorithm for string similarity joins under edit distance (Source Code). https://github.com/kedayuge/MinJoin. Source code accompanying the paper.

[35] Nikolai Karpov and Qin Zhang. 2022. SyncSignature: A Simple, Efficient, Parallelizable Framework for Tree Similarity Joins. *Proc. VLDB Endow.* 16, 2 (2022), 330–342. https://doi.org/10.14778/3565816.3565833

[36] Nikolai Karpov and Qin Zhang. 2022. SyncSignature: A Simple, Efficient, Parallelizable Framework for Tree Similarity Joins (Source Code). https:

//github.com/nkkarpov/syncsignature. Source code accompanying the paper.

[37] Jan Martin Keil. 2019. Efficient Bounded Jaro-Winkler Similarity Based Search. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings (LNI)*, Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klettke, and Holger Meyer (Eds.), Vol. P-289. Gesellschaft für Informatik, Bonn, 205–214. https://doi.org/10.18420/BTW2019-13

[38] Thomas Keller and Patrick Eyerich. 2012. PROST: Probabilistic Planning Based on UCT. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, Lee McCluskey, Brian Charles Williams, José Reinaldo Silva, and Blai Bonet (Eds.). AAAI. http://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4715

[39] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[40] Julien Kloetzer. 2010. Monte-Carlo Opening Books for Amazons. In *Computers and Games - 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers (Lecture Notes in Computer Science)*, H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat (Eds.), Vol. 6515. Springer, 124–135. https://doi.org/10.1007/978-3-642-17928-0_12

[41] Levente Kocsis and Csaba Szepesvári. 2006. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings (Lecture Notes in Computer Science)*, Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou (Eds.), Vol. 4212. Springer, 282–293. https://doi.org/10.1007/11871842_29

[42] Chen Li, Jiaheng Lu, and Yiming Lu. 2008. Efficient Merging and Filtering Algorithms for Approximate String Searches. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen (Eds.). IEEE Computer Society, 257–266. https://doi.org/10.1109/ICDE.2008.4497434

[43] Fei Li, Hongzhi Wang, Jianzhong Li, and Hong Gao. 2013. A survey on tree edit distance lower bound estimation techniques for similarity join on XML data. *SIGMOD Rec.* 42, 4 (2013), 29–39. https://doi.org/10.1145/2590989.2590994

[44] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. 2011. PASS-JOIN: A Partition-based Method for Similarity Joins. *Proc. VLDB Endow.* 5, 3 (2011), 253–264. https://doi.org/10.14778/2078331.2078340

[45] Quanzhong Li, Minglong Shao, Volker Markl, Kevin S. Beyer, Latha S. Colby, and Guy M. Lohman. 2007. Adaptively Reordering Joins during Query Execution. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE Computer Society, 26–35. https://doi.org/10.1109/ICDE.2007.367848

[46] Willi Mann and Nikolaus Augsten. 2014. PEL: Position-Enhanced Length Filter for Set Similarity Joins. In *Proceedings of the 26th GI-Workshop Grundlagen von Datenbanken, Bozen-Bolzano, Italy, October 21st to 24th, 2014 (CEUR Workshop Proceedings)*, Friederike Klan, Günther Specht, and Johann Gamper (Eds.), Vol. 1313. CEUR-WS.org, 89–94. https://ceur-ws.org/Vol-1313/paper_16.pdf

[47] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *Proc. VLDB Endow.* 9, 9 (2016), 636–647. https://doi.org/10.14778/2947618.2947620

[48] Arturas Mazeika, Michael H. Böhlen, Nick Koudas, and Divesh Srivastava. 2007. Estimating the selectivity of approximate string queries. *ACM Trans. Database Syst.* 32, 2 (2007), 12. https://doi.org/10.1145/1242524.1242529

[49] Manoj Muniswamaiah, Tilak Agerwala, and Charles C. Tappert. 2023. Applications of Binary Similarity and Distance Measures. arXiv:2307.00411 [cs.CV] https://arxiv.org/abs/2307.00411

[50] Axel-Cyrille Ngonga Ngomo, Mohamed Ahmed Sherif, Kleanthi Georgala, Mofeed Mohamed Hassan, Kevin Dreßler, Klaus Lyko, Daniel Obraczka, and Tommaso Soru. 2021. LIMES: A Framework for Link Discovery on the Semantic Web. *Künstliche Intell.* 35, 3 (2021), 413–423. https://doi.org/10.1007/S13218-021-00713-X

[51] Santiago Ontañón. 2020. An overview of distance and similarity functions for structured data. *Artif. Intell. Rev.* 53, 7 (2020), 5309–5351. https://doi.org/10.1007/S10462-020-09821-W

[52] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J. Miller, and Divesh Srivastava. 2015. Combining Quantitative and Logical Data Cleaning. *Proc. VLDB Endow.* 9, 4 (2015), 300–311. https://doi.org/10.14778/2856318.2856325

[53] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. 2011. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, 1033–1044. https://doi.org/10.1145/1989323.1989431

[54] Jianbin Qin and Chuan Xiao. 2018. Pigeonring: A Principle for Faster Thresholded Similarity Search. *Proc. VLDB Endow.* 12, 1 (2018), 28–42. https://doi.org/10.14778/3275536.3275539

[55] Chuitian Rong, Wei Lu, Xiaoli Wang, Xiaoyong Du, Yueguo Chen, and Anthony K. H. Tung. 2013. Efficient and Scalable Processing of String Similarity Join. *IEEE Trans. Knowl. Data Eng.* 25, 10 (2013), 2217–2230. https://doi.org/10.1109/TKDE.2012.195

[56] Daniel Schmitt, Thomas Hütter, and Nikolaus Augsten. 2025. *Extensible and Robust Evaluation of Similarity Queries.* Technical Report. University of Salzburg. https://frosch.cosy.sbg.ac.at/dschmitt/fast-similarity-evaluation/-/tree/squashed/paper

[57] Daniel Ulrich Schmitt, Daniel Kocher, Nikolaus Augsten, Willi Mann, and Alexander Miller. 2023. A Two-Level Signature Scheme for Stable Set Similarity Joins. *Proc. VLDB Endow.* 16, 11 (2023), 2686–2698. https://doi.org/10.14778/3611479.3611480

[58] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition.* McGraw-Hill Book Company. https://www.db-book.com/

[59] Yasin N. Silva, Walid G. Aref, Per-Åke Larson, Spencer Pearson, and Mohamed H. Ali. 2013. Similarity queries: their conceptual evaluation, transformations, and processing. *VLDB J.* 22, 3 (2013), 395–420. https://doi.org/10.1007/s00778-012-0296-4

[60] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nat.* 529, 7587 (2016), 484–489. https://doi.org/10.1038/NATURE16961

[61] Ji Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation for Similarity Queries. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1745–1757. https://doi.org/10.1145/3448016.3452790

[62] Ji Sun, Zeyuan Shang, Guoliang Li, Zhifeng Bao, and Dong Deng. 2019. Balance-Aware Distributed String Similarity-Based Query Processing System. *Proc. VLDB Endow.* 12, 9 (2019), 961–974. https://doi.org/10.14778/3329772.3329774

[63] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. 2017. Dima: A Distributed In-Memory Similarity-Based Query Processing System. *Proc. VLDB Endow.* 10, 12 (2017), 1925–1928. https://doi.org/10.14778/3137765.3137810

[64] Maciej Swiechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mandziuk. 2023. Monte Carlo Tree Search: a review of recent modifications and applications. *Artif. Intell. Rev.* 56, 3 (2023), 2497–2562. https://doi.org/10.1007/S10462-022-10228-Y

[65] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. ArnetMiner: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, Ying Li, Bing Liu, and Sunita Sarawagi (Eds.). ACM, 990–998. https://doi.org/10.1145/1401890.1402008

[66] Yu Tang, Yilun Cai, and Nikos Mamoulis. 2015. Scaling Similarity Joins over Tree-Structured Data. *Proc. VLDB Endow.* 8, 11 (2015), 1130–1141. https://doi.org/10.14778/2809974.2809976

[67] Joe Tekli, Richard Chbeir, and Kokou Yétongnon. 2009. An overview on XML similarity: Background, current trends and future directions. *Comput. Sci. Rev.* 3, 3 (2009), 151–173. https://doi.org/10.1016/J.COSREV.2009.03.001

[68] Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. 2021. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *ACM Trans. Database Syst.* 46, 3 (2021), 9:1–9:45. https://doi.org/10.1145/3464389

[69] Carl A. Waldspurger and William E. Weihl. 1994. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI), Monterey, California, USA, November 14-17, 1994.* USENIX Association, 1–11. http://dl.acm.org/citation.cfm?id=1267639

[70] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 85–96. https://doi.org/10.1145/2213836.2213847

[71] Wei Wang, Jianbin Qin, Chuan Xiao, Xuemin Lin, and Heng Tao Shen. 2013. VChunkJoin: An Efficient Algorithm for Edit Similarity Joins. *IEEE Trans. Knowl. Data Eng.* 25, 8 (2013), 1916–1929. https://doi.org/10.1109/TKDE.2012.79

[72] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. 2019. Leveraging set relations in exact and dynamic set similarity join. *VLDB J.* 28, 2 (2019), 267–292. https://doi.org/10.1007/S00778-018-0529-2

[73] Yizao Wang and Sylvain Gelly. 2007. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, CIG 2007, Honolulu, Hawaii, USA, 1-5 April, 2007.* IEEE, 175–182. https://doi.org/10.1109/CIG.2007.368095

[74] Yaoshu Wang, Jianbin Qin, and Wei Wang. 2017. Efficient Approximate Entity Matching Using Jaro-Winkler Distance. In *Web Information Systems Engineering - WISE 2017 - 18th International Conference, Puschino, Russia, October 7-11, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Athman Bouguettaya, Yunjun Gao, Andrey Klimenko, Lu Chen, Xiangliang Zhang, Fedor Dzerzhinskiy, Weijia Jia, Stanislav V. Klimenko, and Qing Li (Eds.), Vol. 10569. Springer, 231–239.

https://doi.org/10.1007/978-3-319-68783-4_16

[75] Yaoshu Wang, Chuan Xiao, Jianbin Qin, Xin Cao, Yifang Sun, Wei Wang, and Makoto Onizuka. 2020. Monotonic Cardinality Estimation of Similarity Selection: A Deep Learning Approach. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1197–1212. https://doi.org/10.1145/3318464.3380570

[76] Manuel Widmoser, Daniel Kocher, Nikolaus Augsten, and Willi Mann. 2023. MetricJoin: Leveraging Metric Properties for Robust Exact Set Similarity Joins. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 1045–1058. https://doi.org/10.1109/ICDE55515.2023.00085

[77] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *Proc. VLDB Endow.* 1, 1 (2008), 933–944. https://doi.org/10.14778/1453856.1453957

[78] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. 2009. Top-k Set Similarity Joins. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng (Eds.). IEEE Computer Society, 916–927. https://doi.org/10.1109/ICDE.2009.111

[79] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM Trans. Database Syst.* 36, 3 (2011), 15:1–15:41. https://doi.org/10.1145/2000824.2000825

[80] Rui Yang, Panos Kalnis, and Anthony K. H. Tung. 2005. Similarity Evaluation on Tree-structured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 754–765. https://doi.org/10.1145/1066157.1066243

[81] Kazuki Yoshizoe, Akihiro Kishimoto, Tomoyuki Kaneko, Haruhiro Yoshimoto, and Yutaka Ishikawa. 2011. Scalable Distributed Monte-Carlo Tree Search. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011, Castell de Cardona, Barcelona, Spain, July 15.16, 2011*, Daniel Borrajo, Maxim Likhachev, and Carlos Linares López (Eds.). AAAI Press, 180–187. https://doi.org/10.1609/SOCS.V2I1.18194

[82] Haoyu Zhang and Qin Zhang. 2017. EmbedJoin: Efficient Edit Similarity Joins via Embeddings. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017*. ACM, 585–594. https://doi.org/10.1145/3097983.3098003

[83] Haoyu Zhang and Qin Zhang. 2019. MinJoin: Efficient Edit Similarity Joins via Local Hash Minima. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 1093–1103. https://doi.org/10.1145/3292500.3330853