A COMPARING DIFFERENT OPERATORS

A.1 Augmentation Operators

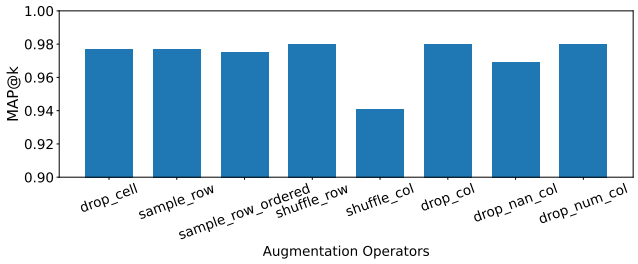


Figure 12: MAP@10 results on SANTOS Small benchmark using different augmentation operators.

To find the most effective augmentation operator used in pre-training (Section 3.2) on the SANTOS Small benchmark, we conduct experiments comparing the MAP@k scores of different op's, shown in Figure 12. Specifically, we experiment with augmentation operators at different table levels, including some of the operators listed in Table 1:

Cell-Level:

- drop_cell: drops a random cell in a column

Row-Level:

- sample_row: samples a random percentage of the rows
- sample_row_ordered: samples random percentage of the rows, while preserving the original order of the rows
- shuffle_row: shuffles the row order

Column-Level:

- shuffle_col: shuffles the column order
- drop_col: drops a random subset of column
- drop_nan_col: drops columns consisting mostly of NaN's
- drop_num_col: drops a random subset of numeric columns

From this ablation study, we find that the column-level operator drop_col leads to the highest MAP@k of 98%, and thus conduct the effectiveness experiments with the drop_col op.

A.2 Sampling Methods

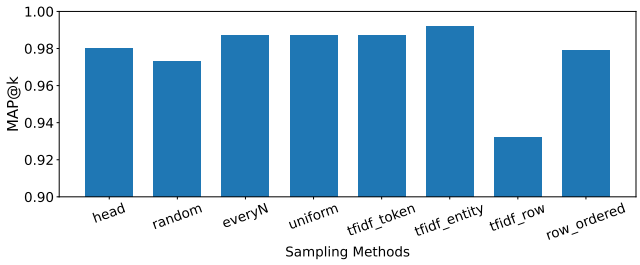


Figure 13: MAP@10 results on SANTOS Small benchmark using different sampling methods.

We also conduct an empirical study comparing different sampling methods to find the method that best preserves the most meaningful tokens in table preprocessing. Specifically, we experiment with the following sampling methods, categorized by the level of the table. Note that all methods preserve the original order of the tokens/cells/rows, while taking unique samples:

Column-Based, Token-Level:

- head: sample first N tokens
- random: randomly sample tokens
- everyN: sample every Nth token
- uniform: sample most frequently-occurring tokens
- tfidf_token: sample tokens with highest TF-IDF
- alphaHead: sample first N tokens sorted alphabetically scores

Column-Based, Cell-Level:

- tfidf_entity: sample cells in a column with highest average TF-IDF scores over its tokens

Row-Level:

- tfidf_row: samples rows with highest average TF-IDF scores over tokens in a row
- row_ordered: sample and serialize tokens in a row

For the design space listed in Section 3.3, we reach the following conclusions from the results shown in Figure 13:

Row-ordered vs. column-ordered: Out of all the sampling methods, the only row-ordered method is "row_ordered" (tfidf_row is column-ordered but selects cells based on the highest average TF-IDF score across the row). The column-ordered methods outperform row_ordered, with the highest column-ordered method tfidf_entity achieving a MAP@k of 99.2% while row_ordered has a MAP@k of 97.9%, thus confirming the original hypothesis.

Token/cell scoring functions: So far we have experimented with simple scoring functions (e.g. head, random), with the most complex scoring function being TF-IDF. However, we can see that the TF-IDF-based methods, specifically tfidf_entity performs the best.

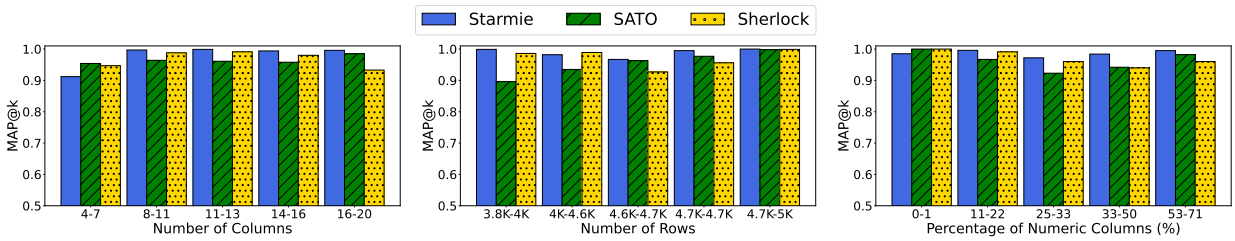
Deterministic vs. non-deterministic: All methods except for "random" are deterministic. Since the best-performing deterministic method, tfidf_entity, outperforms the non-deterministic method "random" (which achieves a MAP@k of 97.3%) we conclude that deterministic methods are more effective.

Row alignment: Methods tfidf_row and row-ordered preserve the row alignment. We can see that column alignment is still more effective, but this design space requires further experimentation.

All in all, this ablation study on the SANTOS Small benchmark shows that the sampling method tfidf_entity performs the best, with a MAP@k of 99.2%. Thus, we conduct our effectiveness experiments on the SANTOS Small benchmark with tfidf_entity as the sampling method.

B IN-DEPTH ANALYSES ON EFFECTIVENESS

Similar to the in-depth analyses conducted for SANTOS Small benchmark, shown in Figure 9, we conduct experiments on both TUS benchmarks to analyze Starmie's performance compared to the baselines SATO and Sherlock. With the same three analyses per benchmark, and the same division of tables into 5 buckets, we first explore the MAP@k results on the TUS Small benchmark. In Figure 14, Starmie again outperforms the baselines and is robust to data



(a) *MAP@k* **of different # Cols** **(b)** *MAP@k* **of different # Rows** **(c)** *MAP@k* **of different % Num. Cols**

Figure 14: In-depth Analyses of Starmie**, SATO, and Sherlock as we vary the number of columns, number of rows, and percentage of numerical columns on the TUS Small benchmark.**

containing large numbers of columns, rows, and high percentage of numeric columns. The baselines also show relatively consistent results across all buckets in the three analyses, which can be attributed to the fact that the TUS Small benchmark is derived from only 10 seed tables, and thus may not exhibit much heterogeneity. This hypothesis requires further analysis.

Conducting the same analyses on the TUS Large benchmark, we see in Figure 15 that Starmie also outperforms the baselines and is consistent across all buckets. On this larger benchmark, even when the baselines' MAP@k performance drops as the number of rows increases or as the percentage of numeric columns increases, Starmie MAP@k remains consistently high, proving again that Starmie performs well across tables of various sizes and columns of different types.

C FULL RESULTS FOR EFFICIENCY EXPERIMENTS

C.1 Efficiency Techniques impact on Performance

Table 8: Efficiency Techniques' impact on query time and performance on the SANTOS labeled benchmark

Method	Technique	MAP@10	P@10	R@10	Q. Time (sec)
Starmie	Linear	0.993	0.984	0.737	96
	Pruning	0.993	0.984	0.737	61
	LSH Index	0.932	0.780	0.580	12
	HNSW Index	0.945	0.810	0.606	4
SATO	Linear	0.878	0.806	0.594	252
	Pruning	0.878	0.806	0.594	125
	LSH Index	0.818	0.712	0.528	89
	HNSW Index	0.730	0.520	0.378	69
Sherlock	Linear	0.782	0.672	0.493	264
	Pruning	0.782	0.672	0.493	145
	LSH Index	0.737	0.612	0.449	100
	HNSW Index	0.705	0.550	0.406	120
SingleCol	Linear	0.891	0.798	0.588	108
	Pruning	0.891	0.798	0.588	100
	LSH Index	0.801	0.538	0.406	11
	HNSW Index	0.803	0.550	0.418	2

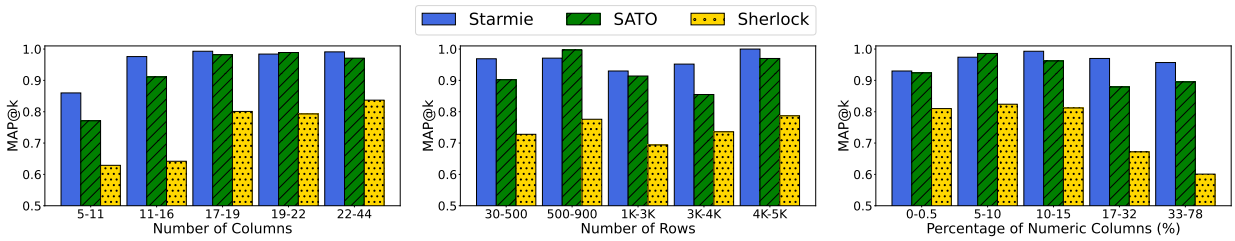
As we experiment with different efficiency techniques in Section 5.3, we also explore their effects on not only the runtimes but also the effectiveness scores. Table 4 explores the different efficiency techniques for the Starmie method, showing that they lead to great speedup while preserving the Starmie performance. In Table 8, we expand on this experiment and apply the efficiency techniques to other embeddings, specifically those of the baselines SATO, Sherlock, and SingleCol. We see that the Pruning technique speeds up the query time by 1.1-2X, while consistently preserving the performance scores perfectly. For indexing techniques, LSH index and HNSW index speed up the query times by 2.6-10X and 2-24X, respectively. Even with the fastest speedup from HNSW index, the baselines SATO and Sherlock are still slower than Starmie while having worse performance scores. As expected, SingleCol is faster as it does not have the cost from the table context. However, even with the fastest query times from the approximation technique HNSW index, Starmie still outperforms all baselines.

C.2 k-Scalability on WDC Benchmark

For the scalability experiments, in Section 5.3 we experiment with the 4 efficiency techniques on Starmie on the SANTOS Large and WDC benchmarks. In Figure 10(a), we show the experiment on SANTOS Large as we increase k from 10 to 60. In Figure 16, we show the same experiment on 1M of the WDC tables. The trends across both figures are similar, with the HNSW index having the fastest query time, followed by LSH index, Pruning, then Linear. However here, HNSW index has a much more impressive performance with the query time remaining around 250 ms as k increases to 60, which is 3000X faster than Linear and 400X faster than LSH index. Thus, Starmie is generally robust in query time as the number of results to return increases, and is sped up the most with HNSW index.

D DETAILS OF COLUMN CLUSTERING

Table 9 shows the full results of column clustering. We use Sherlock, Sato, Starmie, and its single-column version to generate the column embeddings. After obtaining the column embeddings, we construct a similarity graph by adding edges between pairs of columns with similarity above a threshold $\tau = 0.6$. We then cluster the columns by computing their connected components. Note that for fair comparison, we restrict the size of clusters to be around 50 so that different methods generate similar numbers of clusters. We measure the quality of clusters by their purity scores, which measure



(a) *MAP@k* **of different # Cols** **(b)** *MAP@k* **of different # Rows** **(c)** *MAP@k* **of different % Num. Cols**

Figure 15: In-depth Analyses of Starmie**, SATO, and Sherlock as we vary the number of columns, number of rows, and percentage of numerical columns on the TUS Large benchmark.**

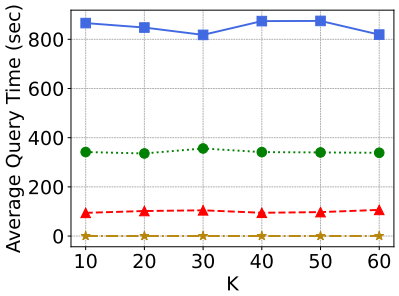


Figure 16: Scalability on 1M WDC tables with varying k's

how likely a column is assigned to a cluster with the same majority semantic type as that of the column. Among the 4 methods, Starmie generates clusters with the highest purity score of 51.19%.

Table 9: Purity scores of clusters by Starmie **vs. Sherlock and Sato.**

	n_clusters	avg. cluster size	Purity (%)
Sherlock	2,395	49.84	30.50
Sato	2,456	48.60	37.36
Starmie	2,297	51.96	51.19
Starmie (SingleCol)	9,252	12.90	20.38

E FULL RESULTS FOR DATA DISCOVERY FOR ML

Table 10 shows the full results of the 25 rating prediction tasks created from 4,130 WDC web tables of ≥50 rows. Each dataset is split into a training and a testing set at a 4:1 ratio. The baseline methods are:

NoJoin: Train a XGBoost model with numeric and textual features from the original table S only. We featurize text attributes using the Sentence Transformers library [41].

Jaccard: Perform an equal left-join with a table that contains a column with the highest Jaccard similarity with any column in the query table. Namely, given a query table $S = \{s_1, \ldots, s_n\}$ of n non-target columns and a data lake $\mathcal{T}$, we join S with the data lake table

$$\underset{T \in \mathcal{T}}{\text{argmax}} \left(\max_{s_i \in S, t_j \in T} (\text{Jaccard}(s_i, t_j)) \right)$$

where $\text{Jaccard}(s_i, t_j)$ is the token-level Jaccard similarity over tokens in query column s_i and data lake column t_j. Note that we exclude "rating" columns from T to avoid any potential label leakage.

Overlap: In this baseline, we simply replace Jaccard similarity from above with the overlap score, i.e., $\text{Overlap}(s_i, t_j) := |\text{tokens}(s_i) \cap \text{tokens}(t_j)|$.

Starmie: For Starmie, we use the learned contextualized embeddings for measuring similarities of columns. Since the embeddings capture the table context of each column, we expect the resulting data tables to be semantically relevant to the query table. More formally, let $\mathcal{M}$ be the learned column encoder, Starmie joins S with the table

$$\underset{T \in \mathcal{T}}{\text{argmax}}(\max_{s_i \in S, t_j \in T} (\cos(\mathcal{M}(s_i), \mathcal{M}(t_j))) + \max_{t_j \in T} (\cos(\mathcal{M}(s_{\text{target}}), \mathcal{M}(t_j)))).$$

Note that we use the second term with the source target column s_{target} (i.e., "Rating") to take into account the similarity between the target column with columns from the data lake table T.

Lastly, an important implementation detail is to make sure that the join result has the exact same number of rows with the query table S. This is done by properly left-joining with the data lake table T. This is done via the pandas DataFrame command:

```
# de-duplicate table T on column t_j
T = T.drop_duplicates(subset=[t_j]).set_index(t_j)
# left-join on the column pair (s_i, t_j)
S.join(T, on=s_i)
```

Table 10: Detailed MSE scores of 25 regressions tasks with different data discovery methods. The Reduction columns measure the improvement of each method against NoJoin.

#row (train+test)	NoJoin	Jaccard	Reduction	Overlap	Reduction	Starmie	Reduction
200	0.0820	0.0885	-0.0790	0.0862	-0.0508	0.0862	-0.0508
200	0.2360	0.2359	0.0003	0.2368	-0.0033	0.2359	0.0003
200	0.0778	0.0653	0.1604	0.0803	-0.0316	0.0653	0.1604
250	0.0008	0.0008	0.0000	0.0008	0.0000	0.0008	0.0000
644	0.0865	0.0880	-0.0174	0.0880	-0.0174	0.0880	-0.0174
533	0.1065	0.1235	-0.1599	0.1235	-0.1599	0.1235	-0.1599
200	0.1269	0.1313	-0.0349	0.1223	0.0365	0.1223	0.0365
200	0.0236	0.0262	-0.1080	0.0232	0.0179	0.0262	-0.1080
535	0.0487	0.0409	0.1586	0.0409	0.1586	0.0409	0.1586
200	0.1598	0.1544	0.0337	0.1195	0.2520	0.1195	0.2520
200	0.0206	0.0214	-0.0389	0.0214	-0.0389	0.0214	-0.0389
529	0.0566	0.0441	0.2208	0.0441	0.2208	0.0441	0.2208
472	0.1731	0.1355	0.2176	0.1355	0.2176	0.1355	0.2176
200	0.0176	0.0197	-0.1178	0.0197	-0.1178	0.0192	-0.0865
200	0.0381	0.0350	0.0824	0.0381	-0.0001	0.0350	0.0824
200	0.0118	0.0097	0.1779	0.0101	0.1420	0.0092	0.2239
200	0.0515	0.0515	0.0000	0.0515	0.0000	0.0515	0.0000
387	0.0662	0.0655	0.0104	0.0685	-0.0344	0.0655	0.0104
434	0.0988	0.0765	0.2250	0.0765	0.2250	0.0765	0.2250
200	0.0177	0.0177	0.0018	0.0177	0.0018	0.0177	0.0018
200	0.1066	0.1066	0.0000	0.0904	0.1522	0.0129	0.8790
200	0.1064	0.0829	0.2210	0.1026	0.0352	0.0829	0.2210
200	0.1875	0.1929	-0.0285	0.1894	-0.0101	0.1894	-0.0101
300	0.0001	0.0001	0.1302	0.0001	-0.0222	0.0001	0.1302
250	0.1488	0.1077	0.2764	0.1152	0.2261	0.1077	0.2764
AVG	0.0820	0.0753	0.0533	0.0748	0.0480	0.0699	0.1050