Search and Join Algorithms for Tables in Data Lakes

by

Erkang Zhu

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Search and Join Algorithms for Tables in Data Lakes

Erkang Zhu
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2019

Data lakes are repositories of data sets stored in their raw formats. Data lakes can be dumping grounds if users cannot find and utilize the data in them. In this thesis, we describe two problems in managing data lakes: searching for tables that can be joined, and auto-generating syntactic transformations for joining tables with join values of different formats. Given a query table and a join column, the first problem is defined as searching for tables that can be joined with the query table on the join column. Our contributions toward solving this problem are twofold: 1) an approximate search index (based on locality sensitive hashing) to support threshold-based search queries – find tables that can join with more than a threshold percentage of the distinct join values in the query table; and 2) an exact search index that supports top-k search queries – find the best k tables that cover the largest number of distinct join values. Both approaches use new data-aware optimizations to provide interactive query performance over real data lakes with millions of tables including many large tables (e.g., millions of rows). Ours is the first approach for searching for joinable tables and we show that it greatly outperforms previous approaches for computing set intersection (used for key-word search and other applications). We also published open source implementations of the joinable table search algorithms and benchmarks created using real data lakes. For the second problem, we propose a technique that generates transformations, without human input, for joining tables with different formats on the join columns. The technique uses a novel approach to pinpoint highly promising joinable row pairs, before using the pairs as input/output examples to perform a greedy search to find a good transformation. The technique scales to tables as large as 10K rows while still maintaining interactive speed. The solutions presented in this thesis make data lakes more searchable and usable, and allow data scientists to be efficient. These experimentally-validated solutions also create an avenue for new data science discoveries that are important in business and government decision-making.

Acknowledgements

I am thankful for the unrelenting support of my supervisor, Professor Renée J. Miller, who has provided much-needed guidance in my progression from a tentative student in the graduate field of data management to a more experienced researcher. She fosters a creative and collaborative environment in the lab. She taught me how to best communicate research ideas in research paper and presentation and spent numerous hours discussing my (sometimes far-fetched) research ideas. Outside the lab, she is a genuine, nice and humorous person and friend.

I would also like to thank my collaborators at the University of Toronto, Professor Ken Q. Pu and my lab mate, Fatemeh Nargesian, for their continuous mentorship and support throughout my five years at the school. Our collaboration started with the development of the LSH Ensemble algorithm in the beginning of my PhD, and continued on today with five significant publications and a Best Demo Award.

Opportunities were afforded to me to participate in collaborations outside of the Data Curation Lab. I am indebted to two fantastic individuals at Microsoft Research, Yeye He and Surajit Chaudhuri. Without their involvement, I would not have had the chance to contribute the Auto-Join algorithm to Azure Machine Learning Data Prep. This exposure helped me better understand the relevant industry problems that exist in data management.

I am fortunate to have worked with Professor Dong Deng of Rutgers University. He introduced me to the field of set similarity search and was a diligent collaborator. Together, we held countless intensive brainstorming sessions during the development of the JOSIE algorithm.

Last but not least, I am grateful to my committee members, Professor Aleksandar Nikolov and Professor Ken Q. Pu, and my reviewers, Professor Felix Naumann at Hasso Plattner Institute and Professor Kelly Lyons at the University of Toronto. They were the earliest readers of this thesis and their contributions, commitments and critiques cannot be understated.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

To begin, we discuss how data lakes differ from traditional databases and data warehouses. We consider the massive and important data being published by governments around the world as an example of a data lake, an open data lake.

1.1 On Data Lakes, Tables, and Open Data

A data lake is a repository of data sets stored in their raw formats. Unlike a traditional data warehouse, a data lake may not have a predefined schema, so the schema of a data set may be defined at read time. This flexibility offers developers and data analysts the freedom to model and query data sets as they need. However, the lack of a fixed schema means users may have limited knowledge about the data sets that are available in a lake.

In data lakes, the table is the most popular data model for data sets. The table data model is also widely used in data science and machine learning. A table typically consists rows of records, all of which have the same set of attributes of a specific data type. A table can also be divided into columns, each of which is an ordered sequence of attributes, each having a specific type of data values. Tables can be stored in many different formats. Traditionally, comma-separated values (CSV), Javascript Object Notation (JSON), and Extensible Markup Language (XML) are the common file formats for tables. Column-oriented formats such as Apache Parquet are becoming popular in Big Data systems. In this thesis, we do not attempt to compare physical file formats used to store the tables; rather, we consider a table as a logical data model.

An important type of data lakes that has emerged in recent years is Open Data portals, which are repositories of data sets published by governments for the purpose of transparency and their potential to spur innovations for better governance and new economic development. Open Data portals typically publish data sets through a catalog software called CKAN, which lists data sets with human-curated metadata under various categories such as Health, Agriculture, and Economics, and which provides limited keyword-based search on titles, tags and description. For the purpose of this thesis, we built a crawler that downloads all data sets in CSV (comma-separated values) format from various CKAN catalogs around the world. The statistics on these various catalogs are shown are shown in Table 1.1.

The numbers in Table 1.1 only reflects CSV files and not other formats that may contain tables because CSV natively encodes a table, while other formats, such as XML and JSON, may require

Table 1.1: Open Data portals with the most CSV tables (October 1, 2018)

Portals	URLs	Number of CSV Files
European Data Portal	www.europeandataportal.eu	258,200
U.S. Open Data	www.data.gov	164,283
U.K. Open Data	www.data.gov.uk	43,109
Canada Open Data	open.canada.ca	36,442
Austria Open Data	www.data.gv.at	21,395
Germany Open Data	www.govdata.de	21,144
Japan Open Data	www.data.go.jp	17,301
Brazil Open Data	dados.rs.gov.br	15,740

schema-dependent transformations to properly identify a table. Nevertheless, even with only the CSV formats, the number of tables is much larger than that in a typical database or data warehouse system and is prohibitively time-consuming for a human to browse through.

1.2 On Joining Tables in Data Lakes

There are many possible operations that a data scientist may want to perform on tables in data lakes in order to obtain interesting analysis. Examples of those operations are transformation, in which data values in a table are converted to different representations (e.g., metric units to the imperial system); union, in which records from different tables are unioned into a common schema (e.g., creating a master rental listing crawled from different websites), and *join*, in which records from one table are merged with records from another table (e.g., merging a table on rental listings with a table on public schools by matching on the district name attribute). In this thesis, we focus on the join operation, which has the power of discovering unknown tables for analysis. We will use Canadian Open Data as an example data lake to demonstrate the power of join.

As an example, an analyst is interested in the firearm-related homicides in Canada. Assume she has access to all the tables downloaded from the Canadian Open Data portal. Through a keyword-based search on titles, she finds the table `Homicide survey, homicides involving firearms` (a real table[1]). The table itself includes the following columns: `Year`, `Type of Firearm`, and `Count`. The analyst could only obtain a time-based trend on different types of firearms and the counts of deaths associated with them.

In order to expand her analysis of firearm homicides, the data scientist may want to find time-based correlations of homicides with other variables in different tables. This option immediately requires looking for other tables with different variables but the same `Year` column (or with a different precision level such as `Year-Month` or `Year-Month-Date`), then applying a join operation to “combine” the tables together, so that in the resulting table every record contains `Year` (or `Year-Month` and so on), `Type of Firearm`, `Count`, and new columns from the other table.

In order to find tables with matching `Year` column, the analyst writes a computer program that scans all tables in the data lake one by one. One interesting table, `Number of police officers`[2], shows up, and it contains a `Year` column and a column for the number of police officers. Both tables are shown in Table 1.2.

[1]https://open.canada.ca/data/en/dataset/be073ee2-a302-4d32-af20-a48f5fbe2e63
[2]https://open.canada.ca/data/en/dataset/07dcaaf5-6a58-4ce3-8876-e30feefef8dd

Table 1.2: Left: `Homicide survey involving firearms`; Right `Number of police officers`

Year	Type of Firearm	Count	...
1974	Total firearms	283	...
1974	Handgun	76	...
1974	Rifle or shotgun	180	...
...	...	...	...

Year	...	Count
1974	...	48075
1975	...	50663
1976	...	51193
...	...	...

Table 1.3: Join result of tables `Homicide survey involving firearms` and `Number of police officers` on `Year`.

Year	Type of Firearm	Count	Num. Police Officers
1974	Total firearms	283	48075
1974	Handgun	76	48075
1974	Rifle or shotgun	180	48075
1974	...	...	...
1975	Total firearms	292	50663
1975	Handgun	88	50663
1975	Rifle or shotgun	183	50663
1975	...	...	...

By joining these two tables on `Year`, the analyst obtains a very interesting result that showcases the correlation between the number of police officers and the number of firearm-related homicides, as shown in Table 1.3. Figure 1.1 shows one possible plot the analyst can generate. She can do further analysis on which type of firearm has the most correlation, and look for more tables to join with to further her analysis.

This case study shows the power of the join operation. However, we do not know how often joinable tables occur in a data lake. One might think there are few joinable tables. Surprisingly, our analysis on the Canadian Open Data shows that there are many: for every table in a random sample, the number of tables joinable on at least 50% of rows are listed in Table 1.4:

From the random sample, one table, `Poll-by-Poll Results 40th General Election`, has columns on vote counts for each candidate and poll location. By joining this table with other tables containing social-economic data related to poll locations, one can potentially discover correlations between social-economic factors and number of votes for each candidate.

Another table from the random sample, the `NSERC Award Data 1993` contains the names of principal investigators on NSERC granst and the amount of the grant. By joining this table with tables containing additional information about principal investigators such as job positions, titles, age, and gender, one can conduct a statistical analysis to find which characteristics of the principal investigators have the highest impact on the funding amount that they receive.

As an additional example, we have conducted an analysis on the correlation between the donations of Candidan companies to political campaigns and the total amount of NSERC funding that they receive. The analysis is performed by joining three tables: `NSERC Award Data 2004`, `NSERC Partners 2004`, and `Campaign Contributions 2000-2004`, and the result is shown in Figure 1.2. This plot shows a correlation that could be studied by a data scientist to understand if there is an underlying causation.

As in our last example, we may wish to perform many joins starting from a query table like the table

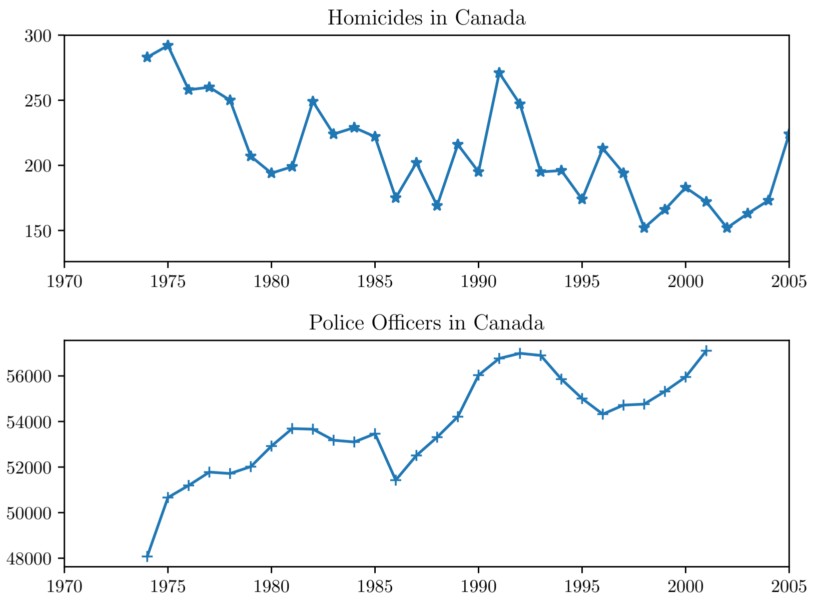


Figure 1.1: An analysis of homicides in Canada involving firearms and number of employed police officers, produced by joining tables `Homicide survey, homicides involving firearms` and `Number of police officers` on years.

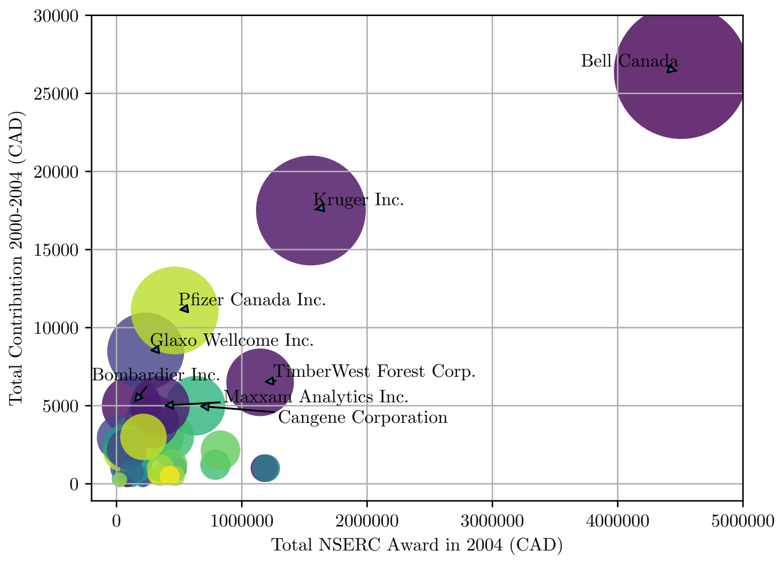


Figure 1.2: An analysis of Canadian corporations' NSERC funding and political campaign contributions, produced by joining three tables: `NSERC Partners 2004`, `NSERC Awards 2004`, and `Campaign Contributions 2000-2004`.

Table 1.4: Sample tables and counts of tables joinable on at least 50% of rows, all from Canada Open Data (March 2017).

Sampled Tables	Num. Joinable Tables	Num. Joinable Columns
International Merchandise Trade 2012-2016	77	323
NSERC Award Data 1993	40	246
Construction Union Wage Rates	14	398
Leisure Time Physical Activity	24	673
Poll-by-Poll Results 40th General Election	196	327

`NSERC Award Data 2004`. By repeatedly expanding our query table through join operations, we can gather more data about the data we are studying. Figure 1.3 illustrates the richness of joinable paths in the Canadian Open Data: we started from a single table (marked as the yellow node), and repeat the operations of finding joinable tables tables (further green nodes), until 4 joins away from the starting table. For this plot, we define joinability as complete containment. The dense connections in Figure 1.3 also show the potential for a data lake such as the Canadian Open Data to be used to augment an existing table that an analyst has already found (or identified).

1.3 On Searching for Joinable Tables

The previous section has shown the potential benefit of joining tables in a data lake. However, we do not yet know how to efficiently find joinable tables when the data lake contains hundreds of thousands of tables. Data lakes of such size exist in some Open Data portals, as we have already seen earlier in Table 1.1. Due to the large size, we want to avoid using linear scan to find joinable tables, and use a sub-linear time search index instead. In addition, some data lakes are growing quickly. Figure 1.4 shows the number of CSV files in some Open Data portals over time. For the European Open Data portal, the number doubles every half-year. Thus, such rapid growth in the number of tables must be handled by an efficient search index.

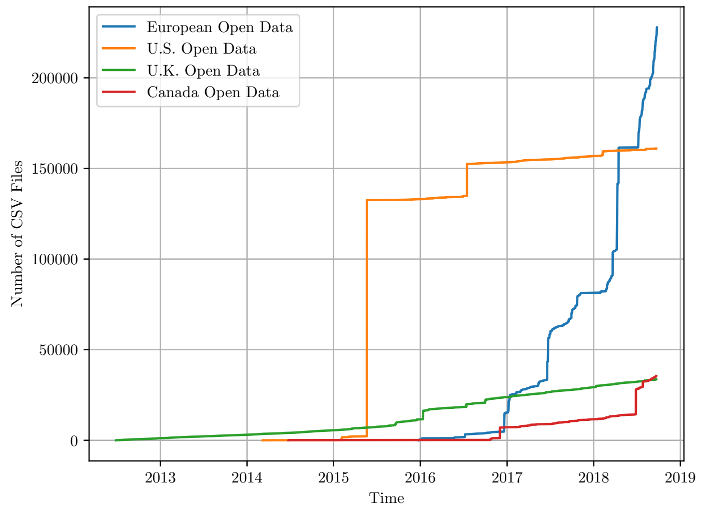


Figure 1.4: The number of tables (CSV only) in Open Data portals over time.

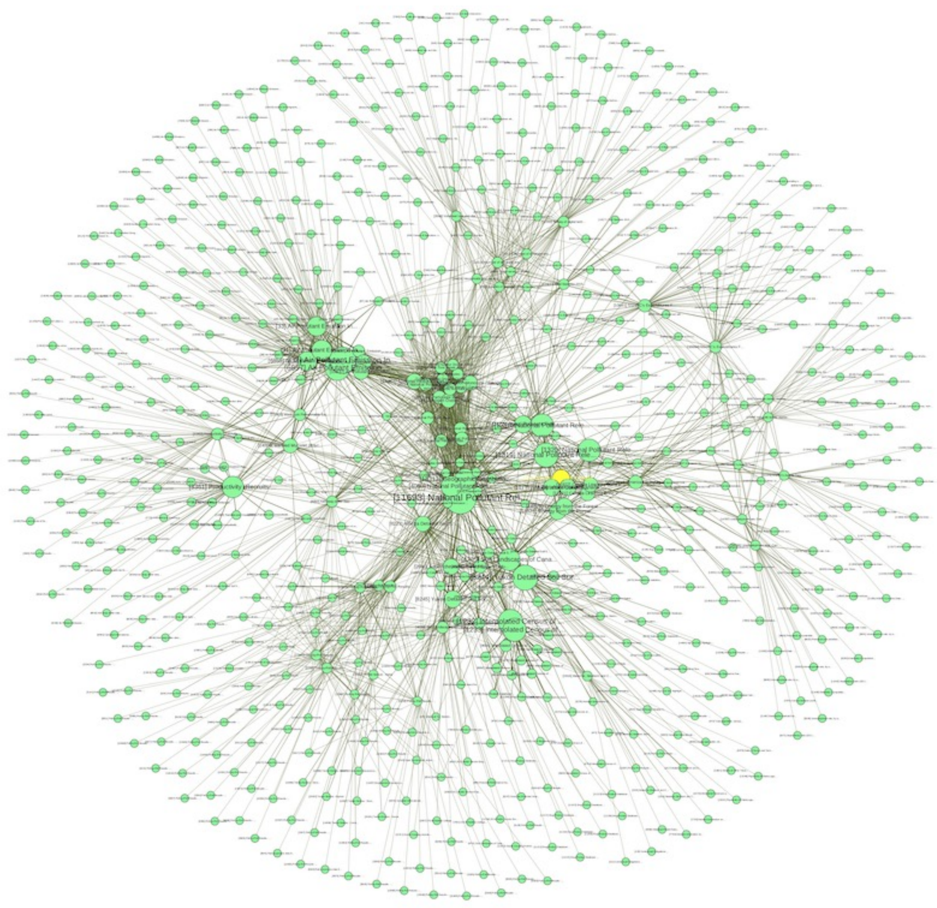


Figure 1.3: The join paths (edges) traversed starting from the yellow table (node) to the 4th degree tables (nodes), within Canada Open Data.

A search index using schema-only information may be suitable for a data warehouse, which usually makes complete (and rich) schema information available to its users. However, for a data lake, a user may have limited or no knowledge of the schema. And the lake is likely too large to browse, as in the case of Open Data portals. Even when the schema information is available to the user, it is likely incomplete or insufficient to identify joinablity among tables. For example, two columns from two different tables with the same name `Year` (or even worse, `Date`) may share few common values, thus cannot produce a join result with enough rows to derive a meaningful analysis.

An alternative to using schema information is to pre-compute all pairs of joinable tables in a data lake, and to build a search index that simply serves the computed pairs as a join graph [62, 26]. This approach provides high performance, since it involves little computation; however, it only works for finding joinable tables for existing tables within the data lake. If a user wants to input a new table as the starting table, he/she still needs to perform an expensive linear scan. In addition, if a user has joined multiple tables from the data lake and performed some data value transformations, the resulting table may not have the same set of joinable tables as the original tables.

In this thesis, to build efficient search indexes for growing data lakes, we propose two approaches: one uses a technique from approximate nearest neighbour search, and the other is an exact search algorithm that provides a major innovation over solutions that use inverted indexes and set similarity search algorithms. These approaches are presented in Chapter 2 and 3 of this thesis. Both of these approaches use only data values, and compute the joinability of tables at search time rather than pre-compute joinable pairs. Thus, they can be used for dynamic data lakes that grow over time.

1.4 On Joins with Transformations

So far, we have been only discussing *equi-joins* – joining rows by exact match of a pair of data values in the join columns. However, equi-join is not the only type of join. As an example of a non-equi-join, we use the table on firearm-related homicides and a table on historical rain falls; they are shown in Table 1.5. Because both tables cover the same date ranges, if they could be joined, then the correlation between historical homicides and rainfalls could be measured.

In order to perform the join, we have to first write a program that creates a new column `Year` in the rain fall table by extracting the year component from the `Year-Month` column. After the join, we can group the joined rows by `Year`, and then take the sum of `Amount` for each group to get the final result.

Table 1.5: Left: `Homicide survey involving firearms`; Right: Synthetic Historical Rainfall Data

Year	Type of Firearm	Count	...
1974	Total firearms	283	...
1974	Handgun	76	...
1974	Rifle or shotgun	180	...
...	...	...	...

Year-Month	...	Amount
1974-01	...	48075
1974-02	...	50663
1974-03	...	51193
...	...	...

Data lake users are often forced to write *syntactic transformation programs* such as extracting the `Year` from `Year-Month` because there are formatting differences among different data sources even for the same class of data values. As an example of storing person names, some data sources use `Firstname Lastname`, while some other sources use `Lastname,Firstname`. Another example is phone numbers: one source starts nation code for phone numbers with `+`, while another source does not have `+` but instead uses brackets around area codes. In addition to differences in formatting for the same value classes, tables may use different but syntactically related unique identifiers. For example, one table extracted from a sign-up sheet uses a person name column, `Firstname Lastname` as the identifier, while another table extracted from a contact list uses an email column `firstname.lastname@example.com`.

Writing a transformation program is a small annoyance if it is only required once and can be used repeatedly. Likely, the written transformation program cannot be applied to other cases of join. The user would have to write a new program for every *ad hoc* join, and this task can become burdensome without contributing to any useful analysis and understanding of the data.

In Chapter 4, we describe a technique for automatically generating transformation programs without user input. This technique can save a lot of valuable time for data scientists working with data lakes, and help them to focus on analytical tasks.

1.5 On Contributions

The primary goal of this thesis is to develop joinable table search algorithms that scale to massive data lakes. Existing work on schema matching [2, 8] and inclusion dependency discovery [3, 52] focused on databases and data warehouses, where the table schema is well-defined and the number of tables is small. In contrast, Open Data data lakes can have hundreds of thousands of tables (see Table 1.1). More recent work (work that follows our own work on joinable table search [74]), finds joinable tables in enterprise data lakes using a pre-computed join graph [26], which can be expensive to maintain for data lakes that grow quickly over time (see Figure 1.4) and are updated frequently. Existing exact and approximate set similarity search algorithms [5, 69, 70, 60] may be used for joinable table search, but these algorithms were not designed for very large tables, large differences in table sizes, and large dictionaries (meaning a large number of distinct data values or vocabulary size) – all of which are common in data lakes.

Thesis Statement: *Joinable table search can be done efficiently at runtime (without a pre-computed join graph) even over data lakes with massive numbers of tables, large dictionaries, and tables of very different sizes (including very large tables).*

We present algorithms for searching for tables that can be joined in massive data lakes, and automatically join them without writing *ad hoc* transformation programs. The algorithms: LSH Ensemble [74], JOSIE [73], and Auto-Join [72] have all been published in the proceedings of the top-tier conferences (VLDB and SIGMOD) and form important parts of the emerging literature on data lake management. Beside algorithmic contributions, we also developed an interactive joinable table search system [75] (using LSH Ensemble as the search index) that won the Best Demo Award at VLDB 2017, released an Open Source Python package[3] for MinHash, LSH, and LSH Ensemble algorithms that has received over 800 stars to date, and published benchmark data sets[4] for evaluating LSH Ensemble and JOSIE.

As data lakes become widely adopted, data management tasks (e.g., searching for tables, joining tables, etc.) will require more hours from data scientists, who already dedicate 80% of their time in data preparation [18]. The algorithms, systems, code, and benchmark data sets presented in this thesis contribute to making data lake management effortless, and encourage future research.

[3]`https://github.com/ekzhu/datasketch`
[4]`https://github.com/ekzhu/set-similarity-search-benchmarks`

Chapter 2

LSH Ensemble: Approximate Joinable Table Search

In this chapter, we present the first approach for efficiently finding joinable tables over data lakes containing a massive number of tables. To begin, we consider what is an appropriate measure for judging whether two tables are "joinable" and whether one table is more joinable to a query table than another (to permit ranking of search results). Give this definition of joinability, we then consider how it can be used in an efficient search algorithm to find joinable tables, when these tables may have variable sizes (varying from a few hundred rows to millions of rows). Our work, known as LSH Ensemble [74], is built upon the theoretical foundation of data sketches and locality sensitive hashing, and our novel search algorithm supports the joinability definition as the relevance measure.

As previously mentioned in Section 1.3, we focus on joinable table search algorithms that use only data values, as schema and metadata may be incomplete or ambiguous in a data lake. In this and the following chapters, we limit the scope further to syntactic techniques – ones that consider data values to be simply string or byte tokens. Other techniques that leverage a semantic understanding of data, by using a knowledge base, entity resolution, or machine learning are also interesting, but syntactic techniques are simpler and more efficient, often used as building blocks for the more complex semantic techniques. For example, in our collaboration with Nargesian et al. on unionable table search [51], we use knowledge base annotations and word embeddings to pre-process tables, but like LSH Ensemble, the search algorithm uses locality sensitive hashing, which is a classic syntactic technique for similarity search.

2.1 Metric for Joinable Tables

Before describing the algorithm, we first define what constitutes a "good" joinable table. In Section 1.2 we have established that the user has an input table, and wants to find tables to join with this input table. We now formalize this problem.

Definition 2.1.1. *A **query table** is a table for which we want to find joinable tables through the search index. A **candidate table** is any table containing a column on which when joined (using an equi-join on a single attribute) with the query table the result is non-empty.*

Table 2.1: Example query table (top) and candidate tables (bottom).

(a) Query table: collaborative NSERC grants between Canadian universities and companies, sampled from the join result of `NSERC Award Data 2004` and `NSERC Partners 2004`, aggregated over amount.

Program	Institution	Partner	Amount
Collaborative Research & Dev...	University of Toronto	Bell Canada	630,683
Research Networks	University of Toronto	Bell Canada	743,818
Research Networks	Université du Québec	Bell Canada	1,500,000
Industrial Research Chairs	Université Laval	Kruger Inc.	566,146
Industrial Research Chairs	York University	MDS SCIEX	300,000
Collaborative Research & Dev...	McMaster University	MDS SCIEX	321,967
Research Networks	Université de Montréal	Corner Brook ...	1,110,000
Research Networks	Université Laval	TimberWest ...	1,120,298
Stratigic Projects - Group	University of British C...	Dofasco Inc.	145,000
Idea to Innovation	McMaster University	Dofasco Inc.	134,450
Collaborative Research & Dev...	University of British C...	Barrick Gold ...	229,930
Research Networks	University of Waterloo	Cangene Corp...	451,613
Synergy Awards	University of Manitoba	Cangene Corp...	25,000
Stratigic Projects - Group	Université de Shebrooke	UMA Engineer...	347,300
Research Networks	Université de Montréal	Pfizer Canada Inc.	440,625

(b) Candidate table: campaign contributions from Bell Canada only.

Recipient	Contributor	Amount
MARTIN,Paul	Bell Canada	1,000
Day,Stockwell	Bell Canada	1,000
Gagliano,Alfonso	Bell Canada	1,000
Copps,Sheila	Bell Canada	1,000
Manley,John	Bell Canada	1,000
GOODALE,RALPH	Bell Canada	1,000
CODERRE,DENIS	Bell Canada	1,000
Clark,Joe	Bell Canada	1,000
Rock,Allan	Bell Canada	1,000
Tobin,Brian	Bell Canada	500

(c) Candidate table: total campaign contributions from all organizations in Table 2.1a except MDS SCIEX.

Contributor	Recipients	Amount
Bell Canada	46	27,885.22
Kruger Inc.	10	20,800
Corner Brook ...	3	3,000
TimberWest ...	2	6,500
Dofasco Inc.	17	3,655
Barrick Gold ...	3	1224.34
Cangene Corp...	1	5,000
UMA Engineer...	3	5,500
Pfizer Canada Inc.	22	15,105

An example of a query table is Table 2.1a, and two possible candidate tables are the two tables Tables 2.1b and 2.1c. In this case, the candidate tables can be joined with the query table by matching records based on data values in `Contributor` column with `Partner` columns. We will call the column in the query table on which we choose to do the join the **query column** and the column in the candidate table on which we do the join the **candidate column** and collectively we call these two columns the **join columns**.

In the example, the query column is `Partner` and the candidate columns are the two `Contributor` columns. As shown in this example, the join columns do not have to be key columns, as in the case of `Contributor`. In addition, an aggregation operation may be useful for non-key columns after (or before) the join operation. In our example, we could do a `GROUP BY` to group the records in Table 2.1b by `Contributor` and sum the `Amount` for each group.

2.1.1 Containment as Relevance Measure

Now, what is a good metric we can use to rank candidate tables? The most obvious metric is the number of records in the output of the join of the query and the candidate. Under this metric, the candidate Table 2.1b generates 30 records, more than the 13 records by Table 2.1c. However, can we say the left candidate is better than the right candidate? No, the right table is likely a better candidate than the left table, for it covers more partners that are present in the query table. A data scientist would likely prefer the right candidate to produce a more comprehensive analysis.

This intuition suggests that the *number of unique values covered* in the query column can be a proxy for measuring the "goodness" of a candidate table. In order to express this quantitatively, we use the following definition of *containment*.

Definition 2.1.2. *For a query column Q and a candidate column X, the* ***containment*** *of X in Q is $\frac{|Q \cap X|}{|Q|}$, where $|Q \cap X|$ is the number of unique values shared between Q and X, and $|Q|$ is the number of unique values in Q.*

In this chapter, we use containment as a relevance measure to rank candidate tables. We consider it a *good* measure because it is not biased when the frequency of data values might be skewed as shown in the previous example (Table 2.1b). The frequent data value `Bell Canada` contributes only a unit score rather than a multiple of its frequency. Thus, this property of containment allows us to better rank candidate tables that are joinable on a column that is not a key (i.e., containing data values that are not unique).

To further demonstrate why containment helps to rank candidate tables joinable on non-key column pairs, we use a new example candidate table, Table 2.2, on campaign contributions to individual recipients. This table contains a list of recipient-to-contributor relationships and associated amounts, and the `Contributor` column is not a key column – its data values are not unique. To join this table with the query table (Table 2.1a), the user may want to perform a `GROUP BY` transformation to compute the total amount of every contributor; count the total number of recipients for every contributor; produce a table that looks like Table 2.1c; and then perform the join operation. Now how do we compare Table 2.1c and Table 2.2? The containment measure tells us that the containment of `Contributor` in Table 2.1c is 90%, while the containment of `Contributor` in Table 2.2 is 50%, regardless of whether the user has performed the `GROUP BY` transformation on Table 2.2. Thus, we can use the containment measure to say that Table 2.1c should be ranked higher (more relevant) than Table 2.2.

Table 2.2: Candidate table: campaign contributions to individuals.

Recipient	Contributor	Amount
MARTIN,Paul	Bell Canada	1,000
Day,Stockwell	Bell Canada	1,000
Gagliano,Alfonso	Bell Canada	1,000
Tobin,Brian	Kruger Inc.	5,000
Chrétien,Jean	Kruger Inc.	5,000
McBreairty,Peter	Croner Brook ...	1,000
McKay,Ian	TimberWest ...	4,000
Tobin,Brian	Pfizer Canada Inc.	1,000

Of course, containment is not the perfect measure in all usage cases – the user may also want as many query table's rows covered as possible by the join operation, or a higher fraction of covered rows. In this respect, the containment measure based on sets (i.e., distinct values) provides a lower bound to the row coverage, and a separate measure based on multi-sets would be required when the query column is not a key.

2.1.2 Jaccard vs. Containment

The definition of containment is similar to that of *Jaccard similarity*, which has a slightly different expression, $\frac{|Q \cap X|}{|Q \cup X|}$, differing by the denominator. Jaccard similarity is widely used in Computer Science, and unlike containment, it is symmetric, making it work nicely with locality sensitive hashing, as we will see in Section 2.3.

Jaccard similarity is not a suitable metric for ranking joinable tables, because it is biased against Q and X that differ greatly in size. This can be demonstrated with the following example.

$$\begin{aligned} Q &= \{\texttt{Ontario}, \texttt{Toronto}\} \\ X_1 &= \{\texttt{Alberta}, \texttt{Ontario}, \texttt{Manitoba}\} \\ X_2 &= \{\texttt{Illinois}, \texttt{Chicago}, \texttt{New York City}, \texttt{New York}, \texttt{Nova Scotia}, \texttt{Halifax}, \\ &\quad \texttt{California}, \texttt{San Francisco}, \texttt{Seattle}, \texttt{Washington}, \texttt{Ontario}, \texttt{Toronto}\} \end{aligned} \tag{2.1}$$

The Jaccard similarity of Q and X_1 is 0.25, while that of Q and X_2 is 0.083. On the other hand, the containment of Q and X_1 is 0.5 while that of Q and X_2 is 1.0. We can see that Jaccard similarity favors columns with a smaller number of distinct values (i.e., set size). Containment, on the contrary, is agnostic to the difference in the set sizes of the columns.

To compare Jaccard similarity with containment using real-world data lakes, we used columns from Canada, US and UK Open Data (see Section: 2.5.2) with set size at least 10. We found 7,132,061 pairs of columns having Jaccard similarity greater than 0.9, while there are 23,044,727 pairs of columns having containment greater than 0.9 – more than 3 times more pairs. Furthermore, all of the Jaccard similarity pairs can be found in the containment pairs. This result demonstrates the usefulness of containment in finding joinable tables, and using Jaccard similarity can miss potentially important results, namely tables that may contain most or all of the query column but are significantly larger. Such tables may still be valuable to a data scientist.

So far, we have only discussed syntactic metrics that are strictly based on data values, and skipped other metrics such as "relevance" judged by the data scientist. Searching for joinable tables under other metrics is certainly important and part of our planned future research, and new techniques will likely use syntactic measures as an essential component (e.g., for first stage filtering). For this thesis, we assume the data scientist would go over the top candidate tables returned by the search index and decide whether to join them.

2.2 The Threshold Containment Search Problem

As discussed in the previous section, we use containment as the metric for joinable table search. Containment is a real number in the range from 0.0 to 1.0, thus it is natural to define the search problem as a threshold search: find candidate columns that contain at least a threshold number of the values in the query column.

Definition 2.2.1 (Threshold Containment Search Problem)**.** *Given a collection of columns* Ω*, the containment function,* $\frac{|Q \cap X|}{|Q|}$*, and a threshold* $t^* \in [0, 1]$ *on containment, find a set of relevant columns from* Ω *defined as*

$$\{X : \frac{|Q \cap X|}{|Q|} \geq t^*, X \in \Omega\} \tag{2.2}$$

Since we will only be dealing with the unique values in every column, we will also use Q and X for the *set of distinct values* in the query and candidate columns respectively.

2.3 Locality Sensitive Hashing for Sets

To scale to massive data lakes containing very large columns (very large sets), our solution to the threshold containment search problem (Definition 2.2.1) is built on the foundation of Minwise Hashing [9] and LSH [36]. We present a brief overview of these two techniques, and a previous work, Asymmetric Minwise Hashing [60], that utilizes Minwise Hashing and LSH for containment search.

2.3.1 Minwise Hashing

Broder [9] proposed a technique for estimating the Jaccard similarity between sets of any size. In this technique, a set is converted into a *MinHash signature* using a set of *minwise hash functions*. For each minwise hash function, its hash value is obtained by using an independently generated hash function that maps all set values to integer hash values and returns the minimum hash value from the set. A MinHash signature for X is then a set of its minimum hash values using a set of different hash functions.

Let h be one such hash function and the minimum hash value of a set X be $h_{\min}(X)$ and Y be $h_{\min}(Y)$. Broder showed that the probability of the two minimum hash values being equal is the Jaccard similarity of X and Y:

$$\Pr[h_{\min}(X) = h_{\min}(Y)] = s(X, Y) \tag{2.3}$$

where $s(X, Y)$ is the Jaccard similarity function. Thus, given the signatures of X and Y, we can obtain an unbiased estimate of the Jaccard similarity by counting the number of *collisions* in the corresponding minimum hash values in two signatures and divide that by the total number of hash values in a single signature.

2.3.2 Locality Sensitive Hashing

The Locality Sensitive Hashing (LSH) index was developed for general approximate nearest neighbor search problems in high-dimensional spaces [36]. An LSH index requires a family of LSH functions. An LSH function is a hash function whose collision probability is high for inputs that are close, and low for inputs that are far-apart. The distance measure must be symmetric. A formal definition of LSH functions can be found in the work of Indyk and Motwani [36].

The minwise hash function $h_{\min}$ belongs to the family of LSH functions for Jaccard distance, as the probability of collision is equal to the Jaccard similarity. For simplicity, we call the LSH index for Jaccard similarity *MinHash LSH*.

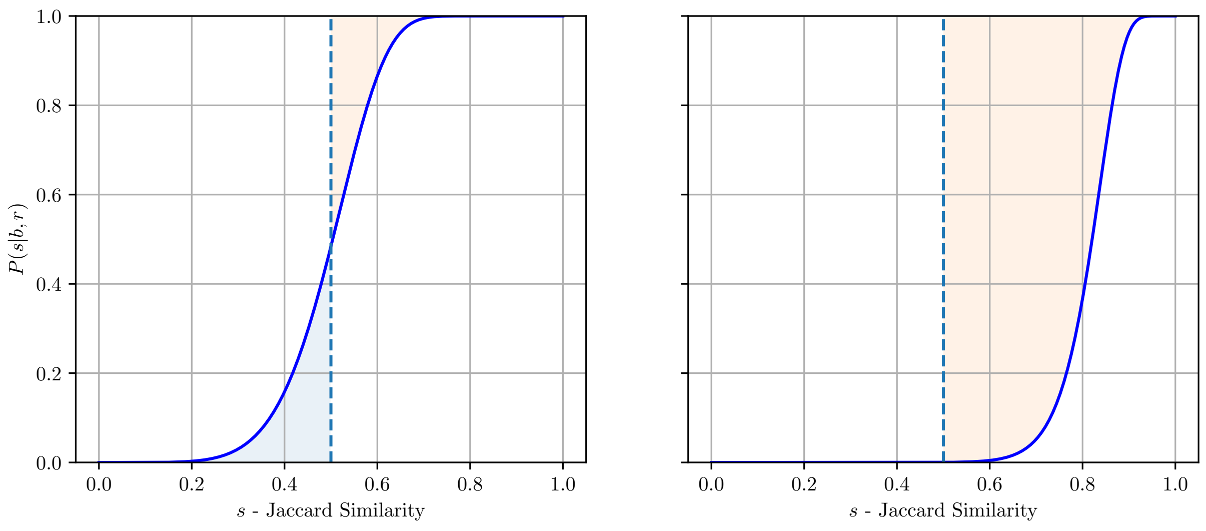


Figure 2.1: The left plot uses $b = 42$ and $r = 6$, and the right plot uses $b = 16$ and $r = 16$; it is clear that with threshold set to 0.5, the left setting is more optimal than the right one.

Next we explain how to build a MinHash LSH index. Assume we have a collection of MinHash signatures of sets generated using the same set of minwise hash functions. The LSH divides each signature into b "bands" of size r. For the i-th band, a function $H_i = (h_{\text{min},1}, h_{\text{min},2}, \ldots, h_{\text{min,r}})$ is defined, which outputs the concatenation of the minimum hash values in that band, namely, the values $h_{\text{min},1}$ to $h_{\text{min,r}}$. The function H_i maps a band in a signature to a bucket, so that signatures that agree on band i are mapped to the same bucket.

Given the signature of a query set, LSH maps the signature to buckets using the same functions $H_1, \ldots, H_b$. The sets whose signatures map to at least one bucket where the query signature is mapped are the *candidates*. These candidates are returned as the query result. Hence, the search time complexity of the LSH only depends on the number of minwise hash functions (or the signature size) and is sub-linear with respect to the number of sets indexed.

The probability of a set being a candidate is a function of the Jaccard similarity s between the query and the candidate. Given the parameters b and r, the probability is:

$$\Pr[s|b, r] = 1 - (1 - s^r)^b \tag{2.4}$$

Given a threshold on Jaccard similarity, we want the sets that meet the threshold to have high probability of becoming candidates, while those do not meet the threshold to have low probability. This can be achieved by adjusting the parameters b and r.

Figure 2.1 demonstrates an example of different settings of b and r affecting the accuracy. The threshold is 0.5, which is shown as the vertical dashed line at the center of both plots. The shaded area left of the threshold line represents the *false positive probability*, which is the likelihood of a set not meeting the threshold but being returned as a candidate; the shaded area right of the threshold line represents *false negative probability*, which is the total likelihood of a set meeting the threshold but not being returned as a candidate. In this case, the left setting is more optimal, as it has lower total false positive and negative probabilities.

LSH has been used widely in applications requiring the fast estimation of the Jaccard measure (or other symmetric similarity measures) over large sets of MinHash signatures [9]. However, until recently, it had not been used with asymmetric measures like containment.

2.3.3 Asymmetric Minwise Hashing

Here we provide a brief overview of Asymmetric Minwise Hashing, an approach based on MinHash LSH, proposed by Shrivastava and Li [60], that supports containment search for sets. We also explain the reason why it is not suitable for containment search over sets where the difference in sizes can be very large. This limitation makes it a poor solution for joinable table search in real data lakes.

Preliminaries

The asymmetric Minwise Hashing technique uses an asymmetric transformation (padding) on sets before creating MinHash signatures and inserting them into an MinHash LSH index. Let M be the largest set we want to index. For every set that is smaller than M, we pad the set with new distinct values to make it of size M. Since the new values have no overlap with the values of the original sets and queries, the containment measure between two sets is the same before and after padding. The effective Jaccard similarity after this asymmetric transformation (padding) can be expressed as:

$$\hat{s}_{M,|Q|}(t) = \frac{t}{\frac{M}{|Q|} + 1 - t} \tag{2.5}$$

In the above equation, the effective Jaccard similarity only depends on the containment t and the query set size $|Q|$ because M is a constant. Furthermore, the effective Jaccard similarity $\hat{s}_{M,|Q|}(t)$ is monotonic with respect to the containment t.

In a MinHash LSH, given a query set, the probability of an indexed set becoming a candidate is determined by its Jaccard similarity with the query set (see Section 2.3.2). Asymmetric Minwise Hashing uses padded sets to build a MinHash LSH index. Because of Equation 2.5, the effective Jaccard similarity of a padded set is monotonic with respect to its containment. Consequently, the probability of an padded indexed set becoming a candidate is also monotonic with respect to its containment. This implies that the effective Jaccard similarity nearest neighbors converges to the containment nearest neighbors, making containment search possible.

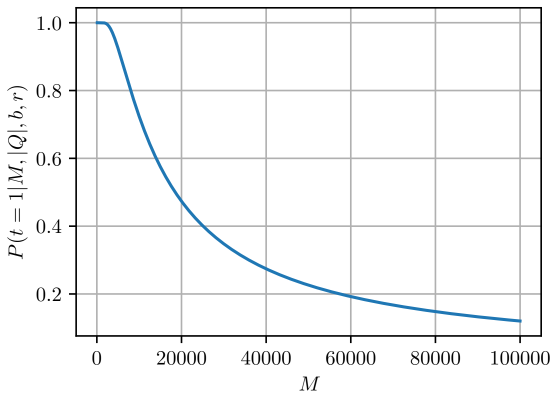
Impact of Spread in Set Sizes

We will show with a simple analysis, how a large spread in set sizes can impact the recall of Asymmetric Minwise Hashing.

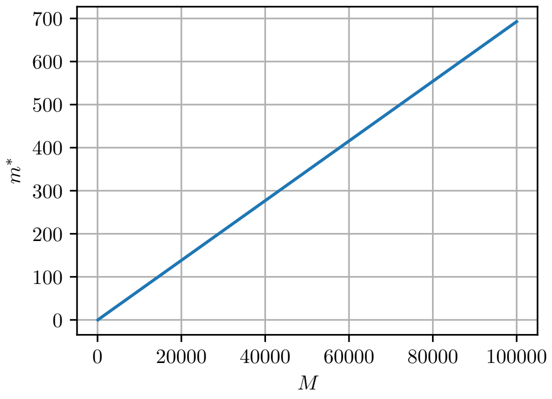
Assume a set has a containment of 1.0 with a query size $|Q|$. Using Equation 2.5 and 2.4, we can derive the probability of the padded set being selected as a candidate by MinHash LSH:

$$\Pr[t = 1.0 | M, |Q|, b, r] = 1 - \left(1 - \left(\frac{|Q|}{M}\right)^r\right)^b \tag{2.6}$$

We can assume that we can tune the LSH to maximize the probability by letting $r = 1$ and $b = 128$ given the number of hash functions is 256. We can further assume $|Q| = 100$ without loss of generality. Figure 2.2a shows the probability of the set being selected decreases very fast as the upper bound of set sizes M increases, even when the MinHash LSH is tuned to maximize the probability.



(a) The return probability of a perfect candidate ($t = 1.0$) using $r = 1$ and $b = 128$



(b) The minimum number of hash functions required to keep the return probability of a perfect candidate ($t = 1.0$) at 0.5

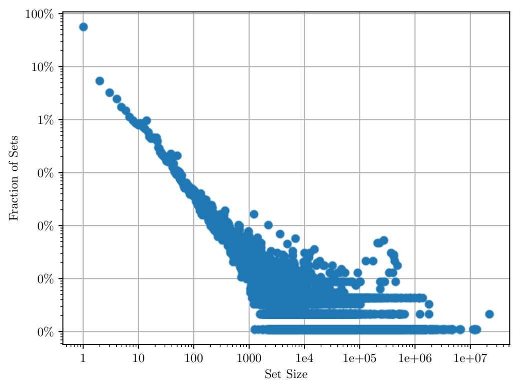
Figure 2.2: Impact of the maximum set size M; $|Q| = 100$

Thus, when the maximum set size M is very large relative to the query set size, the probability for qualifying sets (after padding) being selected as candidates is almost zero, resulting in very low recall. This scenario can happen in practice when the set sizes follow a Zipfian-like distribution, in which M can be much larger than most sets, as in the Open Data and WDC Web Table corpus, shown in Figure 2.3a and 2.3b respectively.

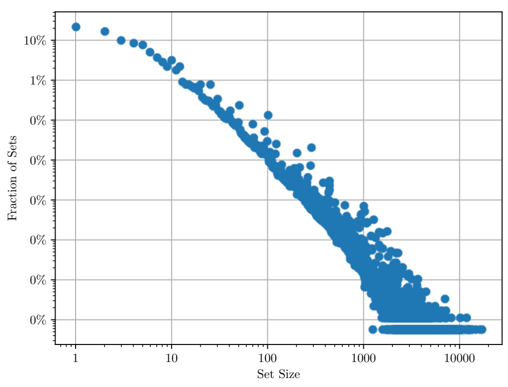
We can still maintain the probability at a certain level by increasing the number of hash functions we use. Let m^* be the minimum number of hash functions required to maintain the return probability of a perfect candidate (i.e., $t = 1.0$) at 0.5. Figure 2.2b shows m^* increases linearly with respect to M. This relationship can be derived using Equation 2.6. In a realistic example, in Canada, US, and UK Open Data lake, the maximum set size is 22 million, this means that if we want a candidate set with containment $t = 1.0$ to have a return probability of at least 0.5 given a query set with size $|Q| = 100$, we need at least 152,492 hash functions. Because the cost of building and searching MinHash LSH index increases with the number of hash functions, having too many hash functions will result in low performance.

2.4 The LSH Ensemble Index

In this section, we describe our contribution, *LSH Ensemble*. In our approach, table columns are represented as sets of distinct values, and the sets are indexed in *two stages*. In the first stage, the sets are partitioned into disjoint partitions based on the set sizes. In the second stage, we construct a MinHash LSH index for *each* partition. We relate containment to Jaccard similarity, and present a cost model that describes the resulting false positive probability if an approximate Jaccard similarity threshold is used to perform containment search. We present a partitioning strategy that optimally minimizes the false positive rate if an ideal Jaccard similarity threshold is used in each partition. We also optimally configure the parameters so that the inherent false positives and false negative errors associated with the LSH indexes are minimized.



(a) The set size distribution of 745,414 sets (non-numeric only) extracted from tables in Canadian, US and UK Open Data portals.



(b) The set size distribution of 163,510,917 sets (non-numeric only) extracted from tables in WDC Web Table 2015 English Relational Corpus.

Figure 2.3: Set size distributions of data lakes

2.4.1 Translating Containment to Jaccard

For the threshold containment search problem, we are given a desired minimal containment as a threshold t^*. On the other hand, MinHash LSH can only be tuned given a Jaccard similarity threshold s^*. Thus, in order to use MinHash LSH for containment search, the containment threshold needs to be translated to a Jaccard similarity threshold.

Consider a set X with set size $|X|$ and query Q with set size $|Q|$. We can translate Jaccard similarity $\frac{|X \cap Q|}{|X \cup Q|}$ and containment $\frac{|X \cap Q|}{|Q|}$ back and forth by the inclusion-exclusion principle:

$$\frac{|X \cap Q|}{|X \cup Q|} = \frac{|X \cap Q|}{|X| + |Q| - |X \cap Q|} = \frac{\frac{|X \cap Q|}{|Q|}}{\frac{|X|}{|Q|} + 1 - \frac{|X \cap Q|}{|Q|}}$$

If $t^* = \frac{|X \cap Q|}{|Q|}$, we can compute the corresponding Jaccard similarity threshold which we denote as $\hat{s}_{|X|,|Q|}(t^*)$. Similarly, if $s^* = \frac{|X \cap Q|}{|X \cup Q|}$, we can compute the corresponding containment threshold which we denote as $\hat{t}_{|X|,|Q|}(s^*)$. The translation functions are given as follows.

$$\begin{aligned} \hat{s}_{|X|,|Q|}(t^*) &= \frac{t^*}{\frac{|X|}{|Q|} + 1 - t^*} \\ \hat{t}_{|X|,|Q|}(s^*) &= \frac{(\frac{|X|}{|Q|} + 1)s^*}{1 + s^*} \end{aligned} \tag{2.7}$$

2.4.2 Jaccard Similarity Filtering for Containment

Notice the translation function $t^* \mapsto s^*$ in Equation 2.7 depends on the set size $|X|$, which is typically not a constant – there are many set sizes in a search index, so we need to approximate $|X|$. We choose to do so in a way that ensures the translation to s^* does not introduce any new false negatives over using t^*. We do this to ensure the index has good recall since false positives can be filtered out, but false negatives mean we may miss good candidate tables in our search. Suppose that X is from a *partitioned* set of sets with sizes in the interval of $[l, u]$, where l and u are the lower and upper set sizes of the

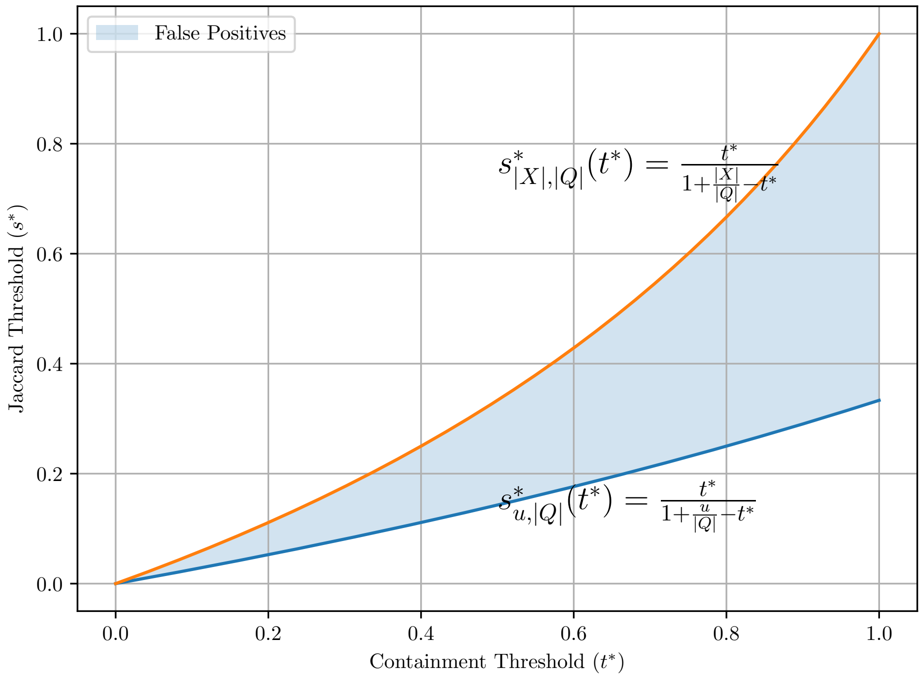


Figure 2.4: The exact (upper curve) and approximate (lower curve) translation functions from containment threshold t^* to Jaccard similarity threshold s^*; and the shaded area corresponding to the false positives introduced by the approximation translation function.

partition. A conservative approximation can be made by using the upper bound u for $|X|$. This ensures that filtering by s^* will not result in any new false negatives.

We define a Jaccard similarity threshold using the upper bound u:

$$s^* = \hat{s}_{u,|Q|}(t^*) = \frac{t^*}{\frac{u}{|Q|} + 1 - t^*} \tag{2.8}$$

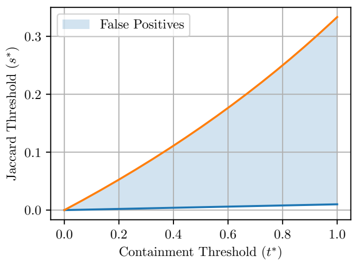
while the exact Jaccard similarity threshold is $\hat{s}_{|X|,|Q|}(t^*)$. Because $u \geq |X|$ and $\hat{s}_{|X|,|Q|}(t^*)$ decreases monotonically with respect to $|X|$, we know $s^* = \hat{s}_{u,|Q|}(t^*) \leq \hat{s}_{|X|,|Q|}(t^*)$. Thus, by using this approximation for s^*, we avoid false negatives introduced by the approximation.

The relationship between the exact translation function $\hat{s}_{|X|,|Q|}(t^*)$ and the approximate translation function $\hat{s}_{u,|Q|}(t^*)$ is illustrated in Figure 2.4. Because $\hat{s}_{u,|Q|}(t^*)$ is a lower bound of $\hat{s}_{|X|,|Q|}(t^*)$ it does not introduce any false negatives. However, the approximate translation function $\hat{s}_{u,|Q|}(t^*)$ introduces false positives: assume a user defined containment threshold $t^* = 0.6$, which is translated to an exact Jaccard similarity threshold of $\hat{s}_{|X|,|Q|}(t^*) = 0.4$, the approximate Jaccard similarity threshold $\hat{s}_{u,|Q|}(t^*) = 0.2$, allowing false positive candidates with Jaccard similarities ranging from 0.2 to 0.4 to pass. Figure 2.4 shows the false positives as the shaded area between the two curves.

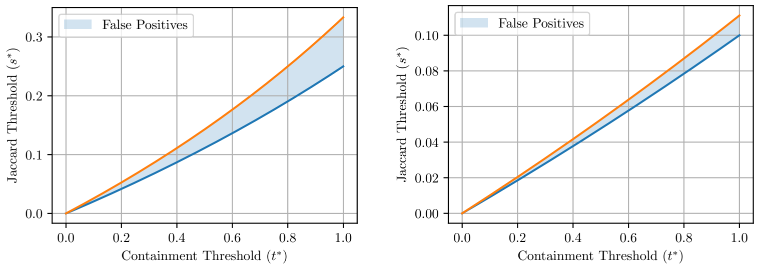
Using the approximate translation function $\hat{s}_{u,|Q|}(t^*)$, we define a search procedure described in Algorithm 1. The query set size $|Q|$ can be computed exactly, or estimated using the MinHash signature [17]. We remark that the choice of s^* will not yield any false negatives, provided that `Jaccard-Search` perfectly filters away all sets with Jaccard similarity less than s^*. Of course, this may not be the case, but our choice of s^* using u guarantees no new false negatives are introduced. We will describe the index

Algorithm 1 Threshold Containment Search

```
function Containment-Search($\mathbf{I}(\mathcal{D})$, MinHash($Q$), $t^*$)    ▷ $\mathbf{I}(\mathcal{D})$: an index of a collection of sets
MinHash($Q$): the MinHash signature of a query set $t^*$: containment threshold
    $l \leftarrow \min\{|X| : X \in \mathcal{D}\},\ u = \max\{|X| : X \in \mathcal{D}\}$
    $s^* = \hat{s}_{u,|Q|}(t^*)$
    $\mathcal{D}_{\text{candidates}} = \texttt{Jaccard-Search}(\mathbf{I}(\mathcal{D}), Q, s^*)$
    return $\mathcal{D}_{\text{candidates}}$
end function
```



(a) False positives when no partitioning is used



(b) False positives of the lower partition (left) and upper partition (right)

Figure 2.5: An example of using no partition versus two partitions

$\mathbf{I}(\mathcal{D})$ and the `Jaccard-Search` shortly.

2.4.3 Partitioning to Reduce False Positives

As discussed in the previous section, the approximate translation function $\hat{s}_{u,|Q|}(t^*)$ introduces false positives. We use a *partitioning* approach, that creates disjoint groups of sets in $\mathcal{D}$, to reduce the false positives caused by the approximation. We can demonstrate the effectiveness of partitioning using an example.

Example 2.4.1. *Let the sets in an index have the following sizes:*

$$\{3, 3, 3, 3, 4, 4, 4, 4, 99, 99, 99, 100, 100, 100\}$$

Without using partitioning, the shaded area corresponding to the false positives (i.e., between the curves for the exact and approximate translation functions, $\hat{s}_{|X|,|Q|}(t^*)$ and $\hat{s}_{u,|Q|}(t^*)$, respectively) are shown in Figure 2.5a. Using partitioning, we can group the sets with sizes $\{3, 3, 3, 3, 4, 4, 4, 4\}$ into one partition, which we call the "lower partition", and the sets with sizes $\{99, 99, 99, 100, 100, 100\}$ into another partition, which we call the "upper partition". Figure 2.5b shows the the false positives of the lower and upper partitions. We can observe a significant decrease in the area corresponding to false positives as we move from no partition (or a monolith partition) to two partitions. This example demonstrates that false positives can be reduced through partitioning.

An LSH Ensemble index partitions the set of *all* sets in $\mathcal{D}$ by their set sizes into disjoint intervals. Let $[l_i, u_i]$ be the lower and upper bounds of the i-th partition, $u_i < l_{i+1}$, and $\mathcal{D}_i = \{X : l_i \leq |X| \leq u_i,\ X \in \mathcal{D}\}$, where $i = 1 \cdots n$. With n partitions, we can search each partition and take the union of the individual query answers. Algorithm 1 provides the search procedure for a *single* partition.

$$\begin{aligned}&\texttt{Partitioned-Containment-Search}(\{\mathbf{I}(\mathcal{D})\}, \text{MinHash}(Q), t^*)\\ =\ &\bigcup_i \texttt{Containment-Search}(\mathbf{I}(\mathcal{D}_i), \text{MinHash}(Q), t^*)\end{aligned}$$

2.4.4 A Cost Model for Containment Search

Let the time complexities of `Containment-Search` and `Jaccard-Search` be denoted by $T_{\text{containment}}$ and $T_{\text{similarity}}$ respectively.

The complexity of Algorithm 1 is given by:

$$T_{\text{containment}} = T_{\text{similarity}} + \Theta(\text{correct result}) + \Theta(N^{\text{FP}}) \tag{2.9}$$

The last two terms are the cost of processing the query result. The value N^{FP} is the total number of false positives from `Jaccard-Search` in all the partitions. In Equation 2.9, we only want to minimize the term $\Theta(N^{\text{FP}})$ to reduce the overall cost.

To minimize the time complexity of the parallel evaluation of `Partitioned-Containment-Search`, we wish to minimize the following cost function by designing the partitioning intervals $[l_i, u_i)$:

$$\text{cost} = N^{\text{FP}} = \sum_{i=1}^{n} N^{\text{FP}}_{l_i,u_i} \tag{2.10}$$

Before we can use this cost model, we must be able to estimate $N^{\text{FP}}_{l,u}$, the number of false positive in a partition. This cost model and FP estimation will allow us to compute the expected cost of a particular partition and develop an optimal partitioning.

2.4.5 Estimation of False Positives

As discussed in Section 2.4.2, by using Jaccard similarity filter with threshold $s^* = \hat{s}_{u,|Q|}(t^*)$ instead of the containment threshold t^*, we incur false positives in the search result of `Jaccard-Search` even if no error is introduced by using MinHash signatures and LSH.

Consider a set X with size $|X|$ in some partition $[l, u]$. Set X would be a false positive if it meets the passing condition of the approximated Jaccard similarity threshold, but actually fails the containment threshold.

Let the containment of the set be t, and the Jaccard similarity be s. The set is a false positive if $t < t^*$ but $s > s^*$. The `Jaccard-Search` filters by s^*, so it is filtering X according to an effective containment threshold of $\hat{t}_{|X|,|Q|}(s^*)$ which we will denote as $t_{|X|}$. Thus, a set is a false positive if its containment t falls in the interval of $[t_{|X|}, t^*]$. In other words, its true containment is below the query threshold t^*, but above the effective threshold $t_{|X|}$.

Proposition 2.4.1. *The effective containment threshold for X is related to the query containment threshold by the following relation.*

$$t_{|X|} = \frac{(|X| + |Q|)t^*}{u + |Q|} \tag{2.11}$$

Given no prior knowledge about the set X, let us assume its containment is uniformly distributed in the interval $[0, 1]$. Thus, we can estimate the probability of a true negative being falsely identified as a candidate.

$$\Pr[X\ is\ \text{FP}] = \frac{t^* - t_{|X|}}{t^*} \tag{2.12}$$

Let $N_{l,u}$ be the expected total number of sets with sizes in the interval $[l, u]$. The expected number of false positives produced by the threshold is given by

$$N^{\text{FP}}_{l,u} = \sum_{|X|\in[l,u]} N_{|X|} \cdot \Pr[X\ is\ \text{FP}] \tag{2.13}$$

where $N_{|X|}$ is the number of sets with size $|X|$.

Given a distribution of set sizes in the interval $[l, u]$, we can evaluate Equation 2.13 further to obtain the following result.

Proposition 2.4.2. *Given a distribution of set sizes $N_{|X|}$ in the set size interval $|X| \in [l, u]$, the upper bound of the number of candidate sets which are false positives is given by*

$$N^{\text{FP}}_{l,u} \le \sum_{|X|\in[l,u]} N_{|X|} \cdot \left(1 - \frac{|X|}{u}\right) \tag{2.14}$$

Proof. (Outline) We need to cover several cases: (1) $t^*q \le l$, (2) $t_l q \le l \le t^*q$ and $t^*q \le u$, (3) $l \le t_l q$ and $t^*q \le u$, (4) $l \le t_l q$ and $t_u q \le u \le t^*q$, and finally (5) $u \le t_u q$. For the first case $t^*q \le l$, we have the probability of a set with size $|X|$ being a false positive being $(t^* - t_{|X|})/t$.

$$\begin{aligned} N^{\text{FP}}_{l,u} &= \sum_{|X|\in[l,u]} N_{|X|} \cdot \frac{t^* - t_{|X|}}{t^*} \\ &= \sum_{|X|\in[l,u]} N_{|X|} \cdot \left(1 - \frac{|X| + |Q|}{u + |Q|}\right) \le \sum_{|X|\in[l,u]} N_{|X|} \cdot \left(1 - \frac{|X|}{u}\right) \end{aligned} \tag{2.15}$$

If $u \gg q$, then the upper bound is a tight upper bound. The other cases (2-5) are proven similarly. See Section 5.4 in the Appendix for the complete proof. □

To summarize, using the distribution of set size within an interval $[l, u]$, we can compute an upper bound of the number of false positives in the interval. This proof can be generalized to other distributions.

2.4.6 Optimal Partitioning

The cost function defined in Equation 2.10, and the estimate of false positive candidate sets in Equation 2.14, allow us to compute the expected cost of a particular partition. In this section, we provide a concrete construction of a minimal cost partitioning.

We denote a partitioning of the sets in $\mathcal{D}$ by $\Pi = \langle [l_i, u_i] \rangle_{i=1}^{n}$ where every pair of consecutive partitions' intervals are *adjacent*, meaning that $u_i < l_{i+1}$ and $\{|X| : u_i < |X| < l_{i+1},\ X \in \mathcal{D}\} = \emptyset$. An optimal partitioning is one that minimizes the total number of false positives over all partitions.

Definition 2.4.1 (The Optimal Partitioning Problem)**.** *A partitioning Π^* is an optimal partitioning if*

$$\Pi^* = \arg\min_{\Pi} \sum_{i=1}^{n} N^{\text{FP}}_{l_i,u_i}$$

Using the distribution of indexed set sizes, we can construct an optimal partitioning Π^* by finding the boundaries. Unfortunately, any Π is query dependent because $N^{\text{FP}}_{l_i,u_i} = N^{\text{FP}}_{l_i,u_i}(q)$ (see Equation 2.15). We cannot afford to repartition $\mathcal{D}$ for each query. Fortunately, with a reasonable assumption, there exists a *query independent* way of partitioning $\mathcal{D}$ near-optimally. We assume that $\max\{|X| : X \in \mathcal{D}\} \gg q$. Namely, $\mathcal{D}$ contains *large* sets relative to the query. An index will most often be used with queries that are much smaller than the maximum set size.

Using the upper bound on $N^{\text{FP}}_{l_i,u_i}$ following Proposition 2.4.2, under the large set assumption, we can see that

$$\text{cost}(\Pi^*) = \sum_{i=1}^{n} N^{\text{FP}}_{l_i,u_i} \approx \sum_{i=1}^{n} \sum_{|X|\in[l_i,u_i]} N_{|X|} \cdot \left(1 - \frac{|X|}{u_i}\right) \tag{2.16}$$

which is query independent. This suggests that we can *approximate* an optimal partitioning Π^* by minimizing the sum of the upper bounds of all partitions.

Definition 2.4.2 (The Approximate Optimal Partitioning Problem). *A partitioning* Π^* *is an approximate optimal partitioning if*

$$\arg\min_{\Pi} \sum_{i=1}^{n} \sum_{|X|\in[l_i,u_i]} N_{|X|} \cdot \left(1 - \frac{|X|}{u_i}\right) \tag{2.17}$$

Because every pair of consecutive partitions are adjacent to each other, and the cost of each partition only depends on itself, we can use dynamic programming to find optimal partition boundaries in $O(|N|^2 \cdot n)$ complexity, where N is the distribution of set sizes, $|N|$ is the number of distinct set sizes, and n is the number of partitions.

Here we provide an overview of the dynamic programming algorithm. We first define the cost of a partition $[l_i, u_i]$ simply as the upper bound of the expected number of false positives, and the cost of multiple consecutive partitions as the sum of all the upper bounds from the partitions. This definition is formalized in Equation 2.18: $\text{Cost}([l_i, u_i])$ is the cost of a single partition starting from l_i and ending at u_i; $\text{Cost}(\Pi^*(l_1, u_i))$ is the cost of an optimal partitioning with i partitions over a domain of set sizes starting from l_1 and ending at u_i.

$$\begin{aligned} \text{Cost}([l_i, u_i]) &= \sum_{|X|\in[l_i,u_i]} N_{|X|} \cdot \left(1 - \frac{|X|}{u_i}\right) \\ \text{Cost}(\Pi^*(l_1, u_i)) &= \sum_{j=1}^{i} \text{Cost}([l_j, u_j]) \end{aligned} \tag{2.18}$$

Given a frequency distribution of set sizes N over a domain of set sizes, $[l_1, u_n]$, and the number of partitions n, the dynamic programming algorithm to find the optimal partitioning $\Pi^*(l_1, u_n)$ works as follow: first compute the cost of all possible single partitions with boundaries $l_j, u_j \in [l_1, u_n]$ and $l_j \le u_j$; then use Equation 2.19 to compute the optimal partitioning $\Pi^*(l_1, u_2)$ of all two-partition cases for all $u_2 \in (l_1, u_n]$; then similarly compute all the three-partition cases for all $u_3 \in (l_2, u_n]$ and so on, finishing at the single n-partition case for $u_n = u_n$.

$$\begin{cases} \text{Cost}(\Pi^*(l_1, u_i)) = \text{Cost}(\Pi^*(l_1, u_{i-1})) + \text{Cost}([l_i, u_i]) & \text{Recursive case} \\ \text{Cost}(\Pi^*(l_1, u_1)) = \text{Cost}([l_1, u_1]) & \text{Base case} \end{cases} \tag{2.19}$$

The complexity of computing the cost of all single partitions is $O(|N|^2)$, and the complexity of computing the cost of all $i > 1$ optimal partitionings, $\Pi^*(l_1, u_i)$, is also $O(|N|^2)$. Since we are performing this operation n times, the total complexity is $O(|N|^2 \cdot n)$.

It is important to note that many real-world domains, set sizes are *sparse*, meaning

$$|N| = |\{|X| : l_1 \leq |X| \leq u_n,\ X \in \mathcal{D}\}| \ll |u_n - l_1| \tag{2.20}$$

Examples are the Open Data lakes, whose set size distributions are shown in Figures 2.3a and 2.3b. The Canada, US and UK Open Data Lake has approximately 14K unique set sizes in the range of $[1, 22075531]$. The WDC Web Tables have approximately 4K unique set sizes in the range of $[1, 17030]$. These statistics imply that, even when the maximum set size is large, the complexity of the dynamic programming algorithm in practice is still computationally feasible. For example, computing the optimal partitioning of 32 partitions on the sets from Canada, US and UK Open Data Lake takes less than 10 minutes using Python on a reasonably modern laptop.

2.4.7 Threshold Containment Search Using Dynamic LSH

In Algorithm 1, we assume the existence of a search algorithm based on a Jaccard similarity threshold for *each* partition $\mathcal{D}_i$. We choose to index the sets in $\mathcal{D}_i$ using a MinHash LSH index with parameters (b, r) where b is the number of bands used by the LSH index, and r is the number of hash values in each band.

Traditionally, MinHash LSH has a fixed (b, r) configuration, and thus has a static Jaccard similarity threshold given by the approximation:

$$s^* \approx (1/b)^{(1/r)} \tag{2.21}$$

One can observe that MinHash LSH only approximates `Jaccard-Search`, and therefore will introduce false positives, in addition to N^{FP} in each partition, and also false negatives. In this section, we describe our method of selecting (b, r) so that the additional error due to the approximation of MinHash LSH is minimized. Since s^* is query dependent, we choose to use a dynamic LSH index, namely LSH Forest [4], so we can vary (b, r) for each query. LSH Forest uses a prefix tree to store the r hash values in each band (see Section 2.3.2), thus the effective value of r can be changed at query time by choosing the maximum depth for traversal in each prefix tree. The parameter b can be varied by simply choosing the number of prefix trees to visit.

We now describe how the parameters (b, r) are selected so that the resulting selectivity agrees with s^*. Using the relationship between Jaccard similarity s and containment t, we express the probability of a set X becoming a candidate in terms of $t = \frac{|X \cap Q|}{|Q|}$ instead of $s = \frac{|X \cap Q|}{|X \cup Q|}$.

$$\Pr[t||X|, |Q|, b, r] = 1 - (1 - s^r)^b = 1 - \left(1 - \left(\frac{t}{\frac{|X|}{|Q|} + 1 - t}\right)^r\right)^b \tag{2.22}$$

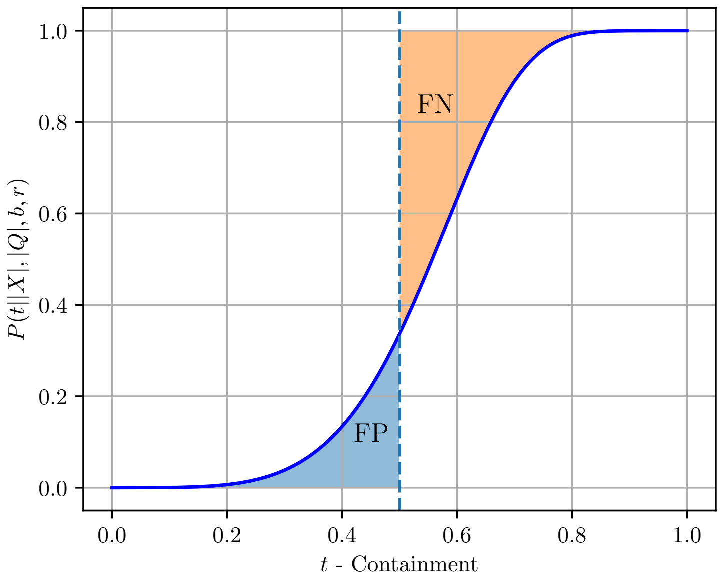


Figure 2.6: $P(t||X|,|Q|,b,r)$ - probability of becoming a candidate with respect to containment, given $|X| = 10$, $|Q| = 5$, $b = 256$, and $r = 4$; containment threshold $t^* = 0.5$ (dashed line)

Figure 2.6 plots the probability (for $t^* = 0.5$), along with the areas corresponding to the false positive (FP) and false negative probabilities (FN) induced by the MinHash LSH approximation. It is important to notice that the false positives here are introduced by the use of `Jaccard-Search` with MinHash LSH, which is different from the false positive introduced by the translation function of the containment threshold to a Jaccard similarity threshold in Section 2.4.5.

Note that t cannot exceed the size ratio $|X|/|Q|$. We can express the probability of X being a false positive, FP, or a false negative, FN, in terms of t^* and $|X|/|Q|$.

$$\mathrm{FP}(|X|,|Q|,t^*,b,r) = \begin{cases} \int_0^{t^*} \Pr[t||X|,|Q|,b,r]dt & \frac{|X|}{|Q|} \geq t^* \\ \int_0^{\frac{|X|}{|Q|}} \Pr[t||X|,|Q|,b,r]dt & \frac{|X|}{|Q|} < t^* \end{cases} \tag{2.23}$$

$$\mathrm{FN}(|X|,|Q|,t^*,b,r) = \begin{cases} \int_{t^*}^{1} 1 - \Pr[t||X|,|Q|,b,r]dt & \frac{|X|}{|Q|} \geq 1 \\ \int_{t^*}^{\frac{|X|}{|Q|}} 1 - \Pr[t||X|,|Q|,b,r]dt & t^* \leq \frac{|X|}{|Q|} < 1 \\ 0 & \frac{|X|}{|Q|} < t^* \end{cases} \tag{2.24}$$

The optimization objective function for tuning LSH is given as:

$$\underset{b,r}{\arg\min}(\mathrm{FN}+\mathrm{FP})(|X|,|Q|,t^*,b,r), \text{such that } 0 < br \leq m \tag{2.25}$$

where m is the number of minwise hash functions. Since $|X|$ is not constant within a partition, we cannot use this objective function for tuning the LSH of the partition. As described in Section 2.4.2, for each partition i, we used u_i to approximate $|X| \in [l_i, u_i]$, and the alternative objective function for tuning we used is:

$$\underset{b,r}{\arg\min}(\mathrm{FN}+\mathrm{FP})(u_i, |Q|, t^*, b, r), \text{such that } 0 < br \leq m \tag{2.26}$$

For some particular value of $|X|$, the b and r found using the alternative objective function would be better if u_i were closer to $|X|$. Formally, for some $\epsilon > 0$, there exists $\delta > 0$ such that when $u_i - |X| < \delta$,

$$(\mathrm{FP}+\mathrm{FN})(|X|, |Q|, t^*, b_p, r_p) - (\mathrm{FP}+\mathrm{FN})(|X|, |Q|, t^*, b_{\mathrm{opt}}, r_{\mathrm{opt}}) < \epsilon$$

where b_p and r_p are the parameters found using the alternative objective function, and b_{opt} and r_{opt} are the optimal parameters computed with the exact objective function.

The computation of (b, r) can be handled offline. Namely, we choose to pre-compute the FP and FN for different combinations of (b, r). At query time, the pre-computed FP and FN are used to optimize the objective function in Equation 2.26. Given that b and r are positive integers and their product must be less than m, the computed values require minimal memory overhead. Finally, we query each partition with the dynamically determined (b, r) using a dynamic LSH index as described by Bawa et al. [4]. In summary, the query evaluation of LSH Ensemble performs dynamic translation of the containment threshold to a per-partition threshold of Jaccard similarity. Then we query each partition with different (b, r) to identify the candidate sets.

2.5 Experiments

In this section, we present the experimental evaluation of LSH Ensemble on real data lakes, and comparison with state-of-the-art approximate and exact algorithms.

2.5.1 Data Lakes

We built a data lake of 215,393 tables by crawling the Canadian[1], U.K.[2] and U.S.[3] open data portals (as of March 2017). We also obtained another data lake of 50,820,165 tables from the WDC Web Tables [40], a public corpus of tables extracted from HTML pages in the Common Crawl.

For each lake, we extracted sets by taking the distinct values in every column of every table. For Canadian, U.K., and U.S. Open Data tables, we also removed all numerical values, as they create casual joins that are not meaningful. We extracted 745,414 sets from the Canadian, U.K., and U.S. Open Data tables, and around 200 million sets from the WDC Web Tables corpus.

2.5.2 Benchmarks

We generated three query benchmarks from the sets extracted from Canadian, U.K., and U.S. open data tables. In order to evaluate performance on different query sizes, each benchmark is a set of queries (sets) selected from one of the three size ranges: 10 to 1k, 10 to 10k, and 10 to 100k. For the rest of this chapter, we will refer to each benchmark using its upper-bound. For example, the benchmark with range 10 to 100k is simply called "100k". For each benchmark, we divide its range into 10 equal-width intervals except for the first interval, which starts from 10. For example, for range 10 to 1k, the intervals

[1]`https://open.canada.ca`
[2]`https://data.gov.uk`
[3]`https://data.gov`

are $[10, 100]$, $[100, 200]$, and all the way to $[900, 1000]$. We then sample 100 sets from each interval using uniform random sampling. Sampling by interval prevents a benchmark that has the same skewed distribution as the repository itself, and heavily biased to sample smaller sets. We want to investigate the effect of query size on the performance of algorithms, thus we need enough queries of all sizes to conduct meaningful comparisons.

For indexed sets, we sampled 74,662 sets, which is 10% of all sets. We did not use all the sets because they would not fit in memory on a single machine.

We also used WDC Web Tables in our scalability experiment (Section 2.5.6), in which we actually index more than 200 million sets using multiple machines.

The benchmarks are published online[4].

2.5.3 Algorithms

We evaluate the accuracy and performance of our LSH Ensemble technique, and compare against several algorithms as baselines:

- `MinHashLSH` is a modified version of the algorithm discussed in Section 2.3.2, proposed by Indyk and Motwani [36]. We modified the original algorithm to support containment search using the approximation technique introduced in Section 2.4.2. It can be considered as `LSHEnsemble` with a single partition.

- `AsymMinHashLSH` is the state-of-the-art LSH algorithm for containment search proposed by Shrivastava and Li [60]. We discussed this algorithm in Section 2.3.3.

- `PrefixFilter` is a state-of-the-art exact similarity search algorithm [5] using an inverted index. We modified the algorithm to support threshold containment search.

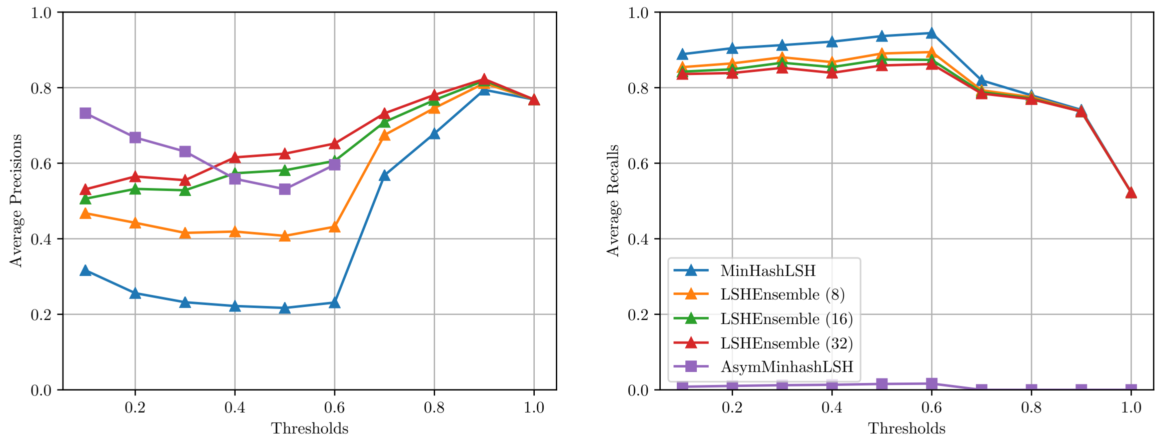
- `LSHEnsemble` is the algorithm presented in this chapter. We used 3 different partitionings containing 8, 16, and 32 partitions labeled as `LSHEnsemble (8)`, `LSHEnsemble (16)` and `LSHEnsemble (32)`.

- `LSHEnsembleParallel` is the same algorithm as `LSHEnsemble`, however, it leverages a multi-core platform to parallelize the querying of the indexes in all partitions. This is especially useful when the complete index is very large. We tested this algorithm only in the scalability experiments in Section 2.5.6.

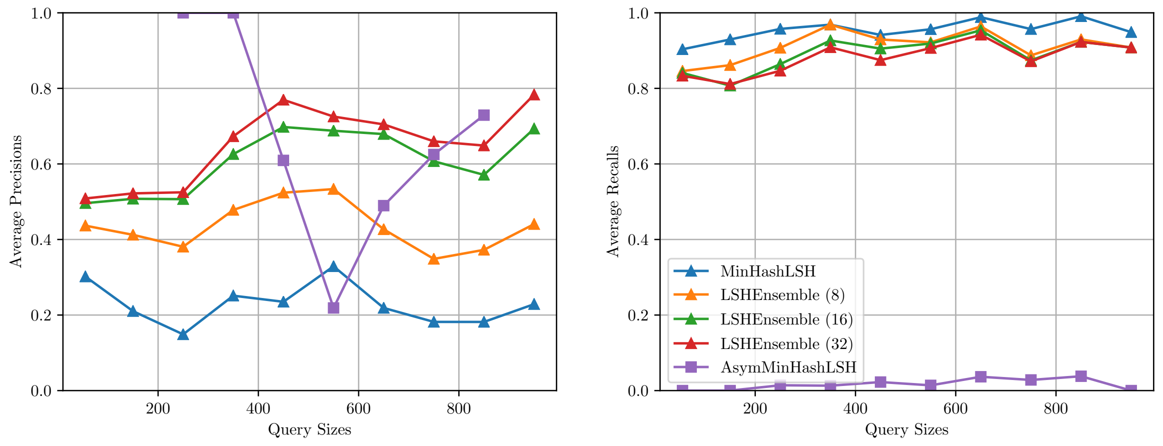
For a fair comparison, `MinHashLSH` and `AsymMinHashLSH` are implemented to use the dynamic LSH algorithm for containment search described in Section 2.4.7, and the upper bound of set sizes is used to translate containment threshold to Jaccard similarity threshold as described in Section 2.4.2.

We have published the implementations `MinHashLSH`, `AsymMinHashLSH` and `LSHEnsemble` in an open source library[5]. We also published our implementations of `PrefixFilter`[6] and `LSHEnsembleParallel`[7]. The benchmark datasets and indexes are stored in memory during experiments.

[4]`https://github.com/ekzhu/set-similarity-search-benchmarks`
[5]`https://github.com/ekzhu/datasketch`
[6]`https://github.com/ekzhu/SetSimilaritySearch`
[7]`https://github.com/ekzhu/lshensemble`



(a) Average precision and recall vs. containment thresholds.



(b) Average precision and recall vs. query size; containment threshold set to 0.6.

Figure 2.7: Precision and recall on 1k query benchmark.

2.5.4 Accuracy

For the accuracy evaluation, we use the set-overlap based definition of precision and recall. Let $\mathcal{D}$ be the set of sets in the index. Given a query set Q and a containment threshold t^*, the ground truth set is defined as $T_{Q,t,\mathcal{D}} = \{X | \frac{|X \cap Q|}{|Q|} \geq t^*, X \in \mathcal{D}\}$. Let $A_{Q,t^*,\mathcal{D}}$ be the set of sets returned by a search algorithm. Precision and recall are defined as follows.

$$Precis. = \frac{|A_{Q,t^*,\mathcal{D}} \cap T_{Q,t^*,\mathcal{D}}|}{|A_{Q,t^*,\mathcal{D}}|}, \tag{2.27}$$

$$Recall = \frac{|A_{Q,t^*,\mathcal{D}} \cap T_{Q,t^*,\mathcal{D}}|}{|T_{Q,t^*,\mathcal{D}}|} \tag{2.28}$$

It is important to note that if the algorithm uses post-processing to remove false positives (i.e., verify exact containment between the query set and the sets returned by the index), the precision will be 1.0. In order to reflect the accuracy of the search index itself, we measure the precision using the unverified results before post-processing.

Figure 2.7, 2.8 and 2.9 plot the precision and recall of the approximate algorithms (`MinHashLSH`, `AsymMinHashLSH`, and `LSHEnsemble`) on the 1k, 10k and 100k benchmarks respectively. We report the average precision and recall for every containment threshold from 0.1 to 1.0 with a step size of 0.1. We also report the average precision and recall for every query size interval in the query benchmarks. Based on these results, the partitioning provides a clear and consistent improvement in precision over the baselines for all thresholds and query sizes, verifying our theoretical analysis of partitioning.

As the number of partitions grows, the precision increases for all containment thresholds and query sizes; however the relative gain diminishes (e.g., from 16 to 32 partitions) as the upper bound of each partition becomes a better approximation of the set sizes of the partition.

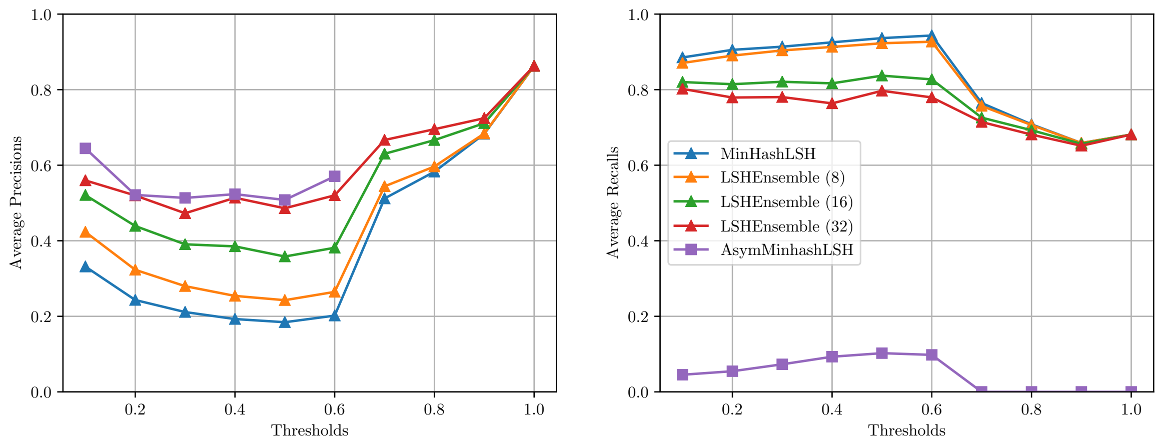
Recall decreases slightly each time the number of partitions doubles. The false negatives are introduced by the MinHash LSH index in each partition, as described in Section 2.4.7. The index tuning becomes less conservative (regarding false negatives) as the upper bound of a partition approaches the actual set sizes. There is a trade-off between partitioning and recall, however, since the approximation always remains conservative, the recall is not affected significantly.

Since the containment threshold changes the optimization constraints for index tuning (see Section 2.4.7) and the parameter space is integer, discontinuity may exist in the optimization objective. This is likely the reason behind the observed variability of precision and recall across different containment thresholds.

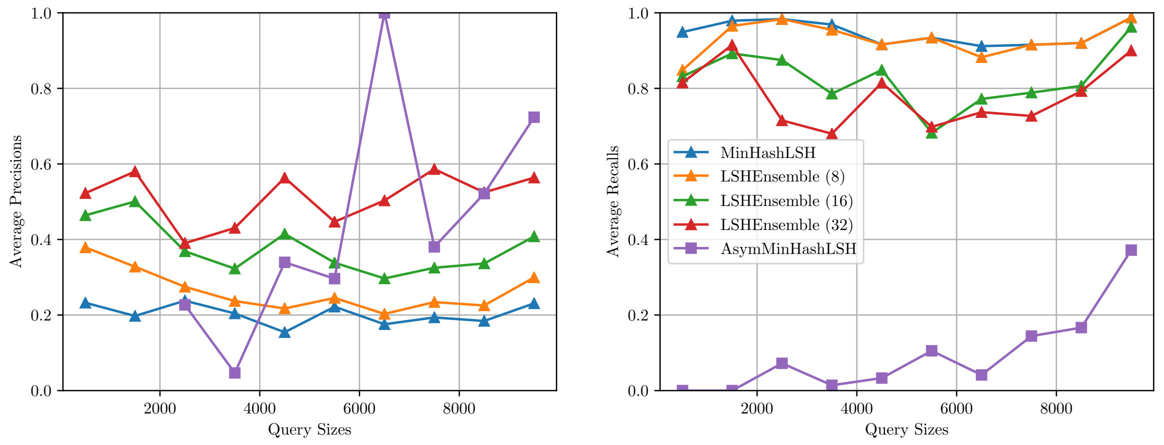
`AsymMinHashLSH` achieved high precision comparable to that of `LSHEnsemble` in all benchmarks, however, performed poorly in recall. Due to padding, some of the hash values in a MinHash signature are from the new padded values. For an indexed set to become a candidate, its hash values from the original set need to have collisions with the hash values in the signature of the query set. Thus, with a finite number of hash functions, the probability of an indexed set becoming a candidate, including sets that should qualify, is lowered. This issue is not significant when the spread in set sizes is small. However, when the spread is large, the amount of padding required becomes very large, making the probability of qualifying sets becoming candidates very low and sometimes nearly zero. This explains the low average recall over the Open Data sets. As the containment threshold increases, sets need to have a greater number of collisions in their hash values. Thus, the probability of finding qualifying sets drops to zero for high thresholds, resulting in a recall of zero. For lower thresholds, recall is better for large queries (e.g., the 100k benchmark at threshold 0.6, Figure 2.9 right plots). This is simply because larger queries lead to larger candidates, which are less affected by padding.

Looking at how many queries actually return something, only 1.6% of the 1k benchmark queries returned non-empty results using `AsymMinHashLSH` for thresholds less than or equal to 0.6. This number is 10% and 30% for the 10k and 100k benchmarks. We consider an empty result having precision equal to 1.0, however, we exclude such results when computing average precision. For joinable table search, a high recall is a necessity. Thus, these results show that `LSHEnsemble` is a better choice for the joinable table search problem over Open Data lakes.

The partitioning assumes the set size of the query is much smaller than the maximum set size of the partitioned index. We investigated whether in practice large query set size influences the effectiveness of the partitioning technique. Figure 2.7b, 2.8b and 2.9b show the precision and recall for queries with different set sizes on the 1k, 10k and 100k benchmarks respectively. For the large queries (e.g., 100k benchmark), the precision is smaller, due to the assumption no longer holding. Still, the precision increases with more partitions, confirming our analysis that partitioning should always increase precision,



(a) Average precision and recall vs. containment thresholds.



(b) Average precision and recall vs. query size; containment threshold set to 0.6.

Figure 2.8: Precision and recall on 10k query benchmark.

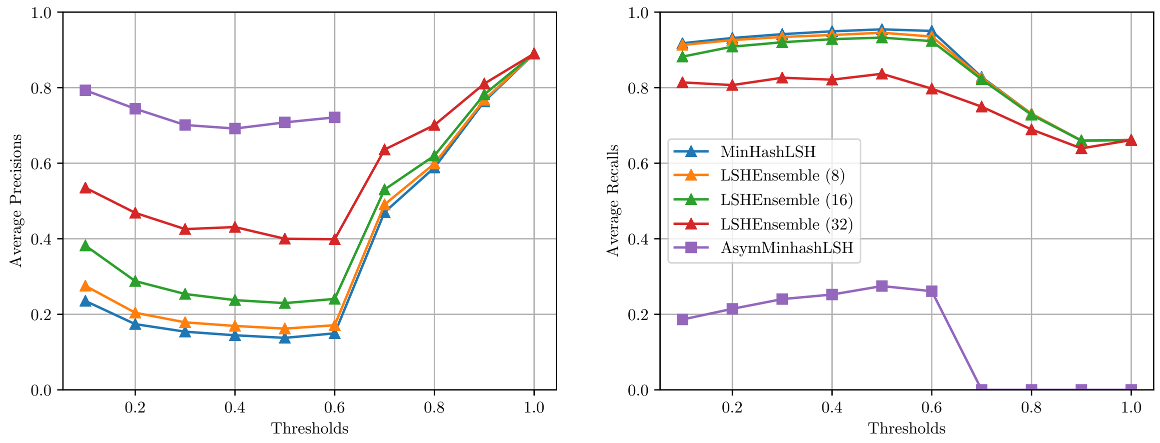
and the recall stays high.

2.5.5 Query Performance

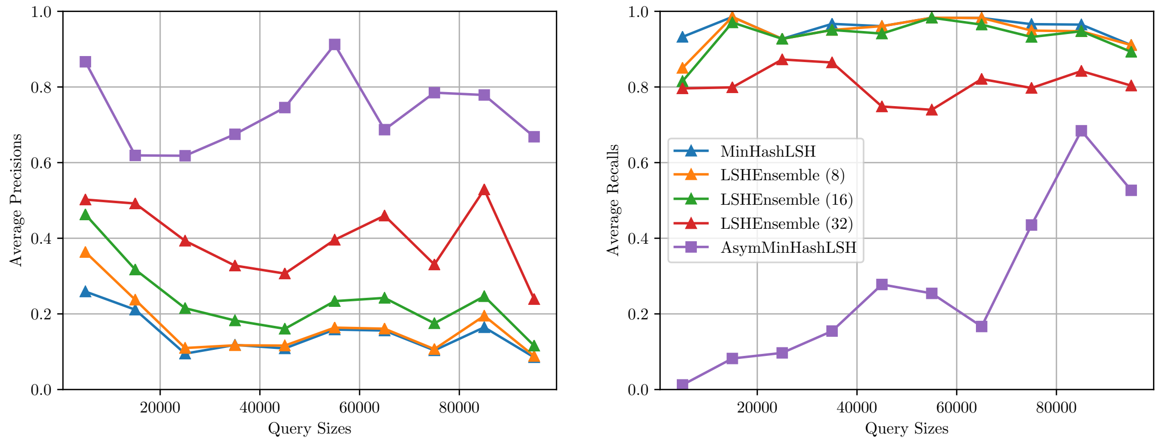
We have compared `LSHEnsemble` against other approximate algorihtms on precision and recall, now we compare it against `PrefixFilter`, a state-of-the-art exact algorithm, on query performance. The experimental results on all benchmarks are shown in Figure 2.10.

On the 1k benchmark (Figure 2.10a), both `LSHEnsemble` and `PrefixFilter`'s query performance improves with increasing threshold, while `LSHEnsemble` constantly out-performs by a factor of 3 to 4. The query time includes the time to verify exact containment of candidates as described in the cost model in Section 2.4.4, thus the query result has 100% precision. The high performance is the most desirable characteristics of `LSHEnsemble` and by extension approximate search algorithms, at the expense of a slight reduction in recall.

On the 10k and 100k benchmarks, the performance advantage of `LSHEnsemble` starts to diminishes a little – the 8-partition becomes slower than `PrefixFilter` at threshold 0.6. This can be explained by the



(a) Average precision and recall vs. containment thresholds.



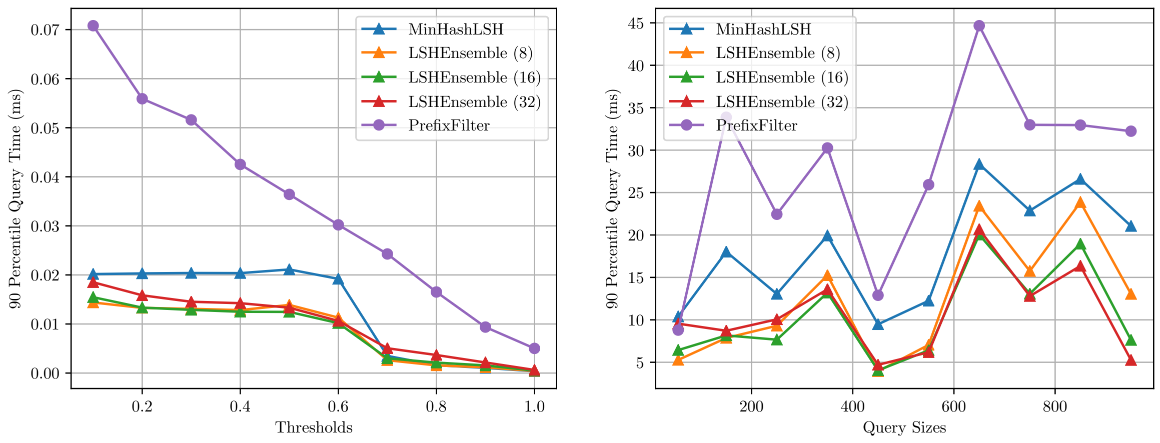
(b) Average precision and recall vs. query size; containment threshold set to 0.6.

Figure 2.9: Precision and recall on 100k query benchmark.

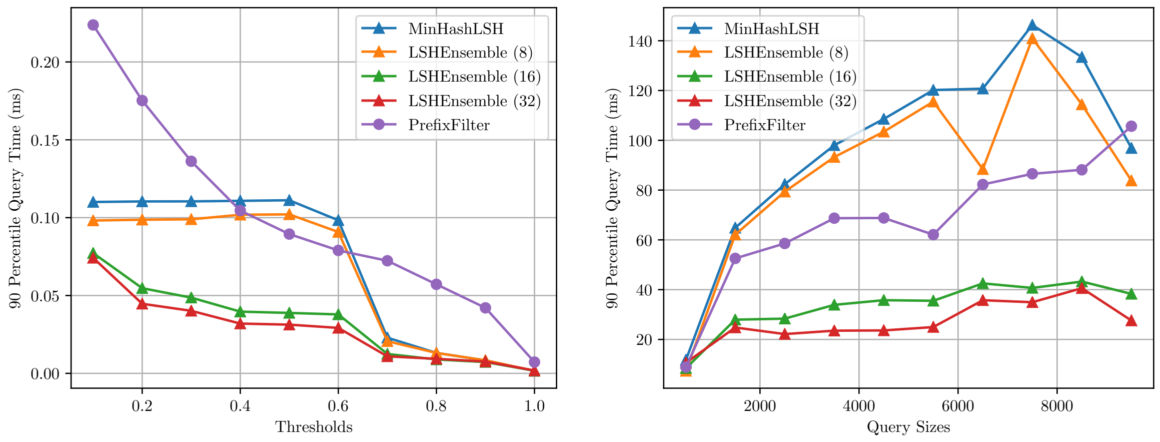
low-precision threshold regions in Figures 2.8 and 2.9 of the accuracy experiments – since the precision is low ($\approx 20\%$), the cost of verifying exact containment and removing false positives becomes higher, slowing down the overall performance. Despite this data point, we can see the benefit of partitioning: the 16 and 32-partition still out-perform `PrefixFilter` by a factor of 2 to 3, because their precision is much higher.

`MinHashLSH`'s query performance stays constant until threshold 0.6, from where it becomes as fast as `LSHEnsemble`. This is also due to the low-precision regions shown in Figures 2.7, 2.8, and 2.9. From this can we see the importance of partitioning: `LSHEnsemble` with 32 partitions is always the fastest, while `MinHashLSH`, having no partition, can be slower than the exact algorithm.

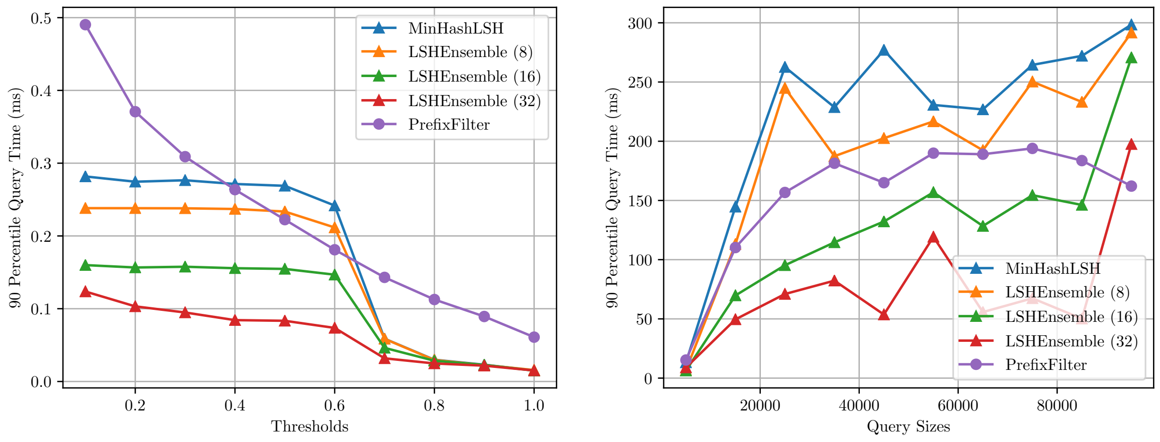
We "zoom in" to threshold 0.6 of all benchmarks and investigate the effect of query size on query performance. For the smaller queries in the 1k benchmark, the query time has high variance and it is unclear if query size has an effect. For the larger queries in the 10k and 100k benchmarks, the query time increases with query size, but the trend is sublinear. Larger queries lead to longer query time because at the same containment threshold, larger queries are more likely to find larger candidates than smaller queries, and larger candidates are more expensive to verify.



(a) 1k query benchmark



(b) 10k query benchmark



(c) 100k query benchmark

Figure 2.10: 90 percentile query time (including post-processing to remove false positives) vs. threshold (left plots) and query size (right plots); threshold is set to 0.6 for right plots.

2.5.6 Scalability

For scalability, we used the English relational subset of the WDC Web Table Corpus 2015, from which we extracted 262,893,406 sets. We selected a random subset of 3,000 sets to use as test queries. Due to the massive number of sets, an index for all sets does not fit in the memory of a single machine. Thus, we divided the sets into 5 equal chunks on 5 machines, and then built an index for each chunk. A query client sends query requests to all indexes in parallel, and the results are unioned.

The first plot in Figure 2.11 shows the indexing cost with respect to the number of sets indexed. The indexing performance of `LSHEnsembleParallel` scales linearly with respect to the number of sets. In addition, because of the parallel implementation, the number of partitions does not affect the indexing cost. The second plot in Figure 2.11 shows the query cost of `LSHEnsembleParallel`. The query cost of `LSHEnsembleParallel` increases with the number of sets in the index, because the number of candidates returned also increases (for a given threshold), and query processing cost includes the cost of outputting the candidates. On the other hand, the query cost increases much slower with more partitions. Again, this is because the precision is improved by partitioning, yielding a smaller selectivity.

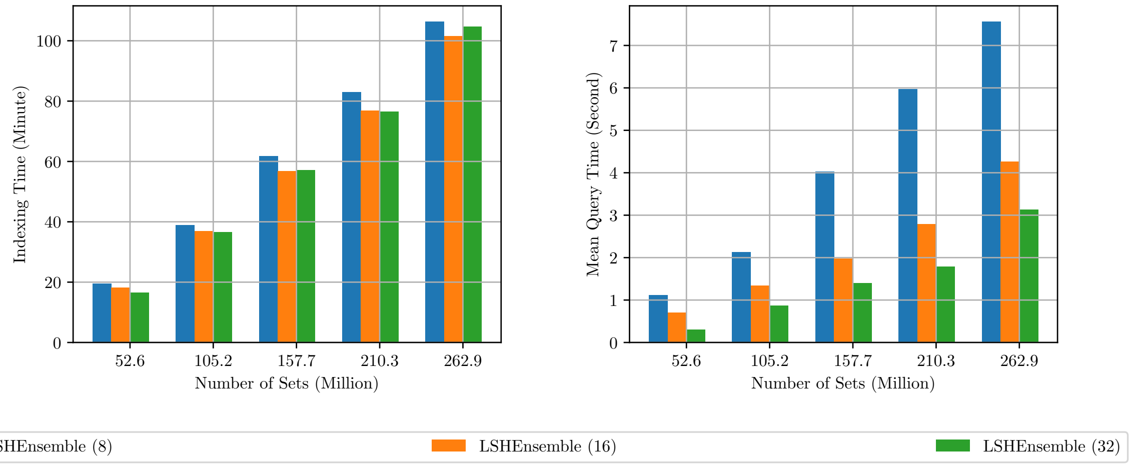


Figure 2.11: Indexing and Mean Query Cost

2.6 Related Work

In this section, we provide a brief survey of previous work related to joinable table search and containment search from different perspectives.

IR Indexes for Keyword Search. Indexes for searching structured datasets given a user-specified keyword query have been extensively studied [16, 34, 38, 53]. This work is similar to containment search as search results should contain the keywords in the query. However, in containment search, the query itself is a column containing a (possibly) large number of data values. To use keyword search to solve containment search, we could generate a keyword for each distinct data value in the column. Such a solution is suitable for small columns. However, we need to handle very large columns containing millions of distinct values. In our solution, the index search time complexity is constant time with respect to the query column size. In addition, the relevance measures in keyword search engines, such as the tf-idf score, consider the frequency of each keyword in all documents. This is different from the relevance measure in containment search, which considers the degree of overlap between the query and the indexed column.

Other LSH Indexes. Various LSH indexes have been developed for distance metrics other than Jaccard, including Euclidean [19], Hamming [36], and Cosine distance [12], among others. These approaches are widely used in areas such as image search and document deduplication. Many such LSH variations require the data to be vectors with fixed dimensions. While it is possible to convert columns into binary vectors, where each unique data value is a dimension, binary vector representation is not practical for domain search at Internet-scale. These indices require prior knowledge of the values in the query domain, hence, they would need to be rebuilt whenever a domain with unseen values is inserted. Thus, LSH indexes, such as MinHash LSH, that do not require a fixed set of domain values are better suited for Internet-scale domain search.

SimHash is an LSH index for Cosine distance [12], and it is applicable to domain search, since it does not prescribe a fixed set of values for all domains. However, it is has been shown that MinHash LSH is more computationally efficient than SimHash [59]. Hence, we use MinHash LSH in our experimental evaluation.

Asymmetric Minwise Hashing. Asymmetric Minwise Hashing by Shrivastava and Li is the state-of-the-art technique for containment search over sets [60] which we discussed in Section 2.3.3. Because we consider a table column as a set of distinct data values, this technique can also be used for joinable table search. While Shrivastava and Li showed that this technique is very useful for containment search over sets extracted from text documents such as emails and news articles, our analysis showed that the asymmetric transformation reduces recall when the set the set sizes can be very different (Section 2.3.3). This analysis was further borne out in our experiments that used Asymmetric Minwise Hashing for join search over a real data lake where its recall was very low, even zero in many experiments (Section 2.5.4).

Schema and Ontology Matching Techniques. Schema matching involves finding attributes in different tables (schemas) that model related information [54]. Schema matching often uses attribute names (which we do not use). Approaches that use only attribute values are sometimes called instance-based. Examples of unsupervised instance-based schema matching techniques are COMA++ [2] and DUMAS [8]. These approaches rely on overlap or similarity of instance values. In order to make instance-based matchers scalable, Duan et al. [24] use MinHash LSH and random projection (simhash) to do type matching based on Jaccard and Cosine similarity. The goal of LSH Ensemble is to match a query column against extremely large number of columns with column sizes varying from a few to millions. LSH Ensemble can be applied in large-scale schema and ontology matching, but to the best of our knowledge, no schema matchers scale to millions of columns. We compare LSH Ensemble with MinHash LSH (used by Duan et al. [24]) experimentally in Section 2.5, and show that LSH Ensemble provides better performance over data lakes.

Inclusion Dependency Discovery Techniques. Given two tables and two lists of attributes, $R(X)$ and $S(Y)$, an inclusion dependency (IND) holds if every tuple in $R(X)$ also appears in $S(Y)$. If the attribute lists contain a single attribute, the dependency is a unary inclusion dependency. An important challenge in IND discovery is managing the exponential problem of considering all combinations of attributes. Some approaches have a first phase that searches for all unary inclusion dependencies and compute single attribute containment as we do in join search. Unary IND discovery can require exact containment or some approaches, which find *approximate INDs*, require the containment to be above a threshold [48]. In contrast to our work however, these approaches find all INDs or all approximate INDs within a relational database and typically scale to only a small number of tables (and attributes) and small numbers of unique attribute values. Examples of more scalable algorithms for inclusion dependency discovery are SPIDER [3], which takes advantages of parallel computation, and BINDER [52], which uses a divide-and-conquer approach. Tschirschnitz et al. [62] also studied scaling up inclusion dependency discovery over many tables, such as the WDC Web Table corpus. Notably, these more scalable algorithms only find exact (not approximate) INDs. And importantly, they are batch algorithms that find all dependencies among a set of tables, and are not applicable to the search problem when the query table is not in this set.

Methods for Table Search. Table search is the problem of finding join candidates in a repository of relational tables. Semantic-based table search approaches enrich tables with ontology-based semantic annotations [63, 44]. Semantic-based table search is complementary to our approach which uses syntactic set containment. Some table search approaches rely on table context and metadata in addition to table content to do search [41, 71]. For instance, InfoGather uses a matching measure consisting of features such as the similarity of text around tables in Web pages, the similarity of each table to its own context, the similarity of the URL of the web page containing each table, and the number of overlapping tuples in tables [71]. Open data sets often lack text context, meaningful descriptions, and reliable schema information. Our approach relies solely on the data values of columns in tackling the joinable table search problem. LSH Ensemble can be used as a component in table search engines when additional information is available.

2.7 Summary

We proposed the *threshold containment search problem* where the goal is to find sets that maximally contain a query set from tables in data lakes. We presented *LSH Ensemble*, a new index structure based on MinHash LSH, as a solution to the threshold containment search problem.

By means of partitioning, we show how we can efficiently perform set containment-based queries even on hundreds of millions of sets with highly skewed distributions. We constructed a cost model that describes the precision of LSH Ensemble within a given partition. We show that for any data distribution, there exists an optimal partitioning scheme that minimizes the expected number of false positives, and we present a dynamic programming algorithm to compute the optimal partitioning efficiently.

We have conducted extensive evaluation of LSH Ensemble on Open Data tables and the entire English relational WDC Web Table Corpus. Our solution is able to sustain query times of few seconds even over 262 million sets.

Chapter 3

JOSIE: Exact Top-K Joinable Table Search

In this chapter, we present JOSIE, an algorithm for exact top-k overlap set similarity search that allows for efficient joinable table search in data lakes. This algorithm solves a different problem from LSH Ensemble: it finds a ranked list of k most relevant candidate tables with the highest containment, and its search result is exact. JOSIE completely out performs the state-of-the-art exact techniques, while being closely on par with LSH Ensemble (when adapted to the top-k problem) and even faster in some cases, in experiments using a real-world Open Data data lake and the WDC Web Table corpus. JOSIE's high performance across different data lakes is due to its adaptive minimization of the read cost of sets and posting lists. This cost minimization is especially advantageous when both the quantity and sizes of sets and posting lists are large, as is typical in data lakes. LSH Ensemble remains the fastest solution when k is large, but the large number of scenarios where JOSIE can out perform an approximate technique like LSH Ensemble is an important, and perhaps surprising, contribution of this thesis. The algorithm and experimental evaluation presented in this chapter are published work [73].

3.1 The Top-K Overlap Set Similarity Search Problem

As discussed in the introduction, a search engine for joinable tables asks a user to input a table and specify a query column, and returns tables that can be joined with the input table on the query column. In the last chapter, we framed the joinable table search problem as a threshold containment search problem for sets, in which a user provides a set as the query and a containment threshold, the index returns candidates with containment scores greater than or equal to the threshold. Containment measures how many distinct values in the query column can be joined with the candidate column.

Threshold-based search functionality is useful when the user understands its meaning and knows what is a good threshold to choose that results in not too many or too few results. However, this assumption may not always hold, especially when the user is dealing with an unfamiliar data lake.

An alternative search problem is top-k: the user provides a set, and instead of a threshold, he/she provides an integer k, which specifies the maximum number of *ranked* candidates to return. The returned candidates must be the ones with the highest containment. The user does not need any prior knowledge to specify k. For example, a small value (e.g., 10) may be sufficient to get an understanding for what is

Table 3.1: Characteristics of sets derived from three data lakes in comparison with other datasets used in previous work.

	#Sets	MaxSize	AvgSize	#UniqTokens
OpenData	745,414	22,075,531	1,540	562,320,456
WebTable	163,510,917	17,030	10	184,644,583
Enterprise	2,032	859,765	4,011	3,902,604
AOL	10,054,184	245	3	3,900,000
ENRON	517,422	3,162	135	1,100,000
DBLP	100,000	1,625	86	6,864

available in the data lake.

Recall containment of a candidate X given query set Q is $\frac{|Q \cap X|}{|Q|}$. Since for a given instance of a search task, the Q is fixed, the returned candidates can be ranked simply using the nominator $|Q \cap X|$, which is called the *intersection size*[1] between Q and X.

Let us define the top-k overlap set similarity search problem in the context of finding joinable tables. We take all columns of all tables in a repository of tables, convert every column into a set of distinct values, and call the big collection of sets Ω. Let Q be the set of distinct values in a user specified column. The top-k overlap set similarity search problem can be defined as follows.

Definition 3.1.1 (Top-k overlap set similarity search problem)**.** *Given a set Q (the query), and a collection of sets Ω, find a sub-collection ω of at most k sets such that:*

1. $|Q \cap X| > 0,\ X \in \omega,$ *and*
2. $\min\{|Q \cap X|,\ X \in \omega\} \geq |Q \cap Y|,\ Y \notin \omega,\ Y \in \Omega$

The key here is that the k-th (or the last) set in the result ranked by set intersection size with the query, has the same or larger intersection size as any other set that is not in the result. As we will explain in the next section, the k-th intersection size in the result can be used as a threshold to prune unseen sets.

Overlap set similarity search is an instance of the *set similarity search* problems that have been studied extensively. Notably, solutions for this problem have focused on relatively small sets such as keywords and titles [42, 69, 6, 65, 22, 66]. The state-of-the-art solutions rely on *inverted indexes*, which contain *posting lists* that map every distinct value in a set (or token) to a list of indexed sets that contain it. The solutions involve either 1) reading all posting lists in the query set to find candidates, and then ranking them based on the number of times they appear, or 2) reading a subset of posting lists to locate candidates, and then reading the candidates to find the final ranked list of sets. Most of these solutions assume the inverted index is relatively small.

The characteristics of modern data lakes take this classic problem to the next level. Data lakes may have large set sizes and huge dictionaries (number of distinct values) as shown in Table 3.1. In this table, we include three data lakes. The first is a lake of Open Data that was used in our evaluation of LSH Ensemble (Section 2.5.1), It contains the non-numeric attributes from 215,393 tables from U.S., U.K., and Canadian Open Data portals as of March 2017. The second is the subset of the WDC Web

[1]Past work in set similarity search sometimes called this "overlap similarity" [70, 65]. We find the term "intersection size" to be less ambiguous. In this thesis, we use the latter, but keep the common name "overlap set similarity search".

Table Corpus [40] which was also used in our LSH Ensemble evaluation. The third are statistics from a private enterprise data lake from a large organization (obtained from Deng et al. [20]). The last three lines report not data lakes, but rather typical datasets used in previous work for set similarity search [69, 6, 65] (numbers from Mann et al. [46]). Notice that the previous work has used data with much smaller set sizes and far fewer unique values.

These differences have significant implications. First, due to the huge number of unique tokens, an inverted index will use a lot of space – approximately 100 GB in PostgreSQL. So, memory management becomes an issue for the index and reading the whole index (all posting lists) becomes infeasible. Second, sets are large as well so again memory management is an issue and it is important to limit the number of sets read. Consequently, the cost of reading a posting list or a set in data lakes will be more expensive than considered in previous work, and an approach that reads all the posting lists of a large query set, or reads too many candidates, will not be feasible.

It is important to note that LSH Ensemble can scale to very large sets. Since the result of the top-k overlap set similarity search is the same as top-k containment set similarity search, it can also be easily modified to solve the top-k problem, by trying decreasing containment thresholds, and ranking the candidates in the search result.

In this chapter, we will present a new exact algorithm that we call JOSIE. Surprisingly, for reasonable k, the new algorithm out-performs LSH Ensemble for query sizes up to 10K. It also out-performs the state-of-the-art exact algorithms by a factor of 3.

3.2 Inverted Index, Prefix Filter, and Position Filter

In this section, we lay down the foundation on which our exact top-k algorithm is built. This includes the inverted index, prefix filter, and position filter. We also introduce two baseline algorithms that solve the top-k overlap set similarity search problem, and methods for calculating the cost of these algorithms.

3.2.1 Inverted Index and Dictionary

A lot of work on exact set similarity search uses basic techniques from Information Retrieval, among which the most common one is the inverted index [47]. It has been widely used in search engines for documents and web pages. Here, we describe inverted indexes in the context of overlap set similarity search. For a collection of sets $X_1, X_2, ... \in \Omega$, we extract the *tokens* (i.e., values) $x_1, x_2, ...$ from all the sets, and for each token, we build a *posting list* of pointers to the sets that contain the token, and these posting lists together form an inverted index.

Example 3.2.1. *For example, the following three sets can be used to build an inverted index containing six posting lists.*

$$\begin{aligned} X_1 &= \{x_1, x_2, x_3\} \\ X_2 &= \{x_3, x_4, x_5\} \\ X_3 &= \{x_2, x_4, x_6\} \end{aligned} \quad \Longrightarrow \quad \begin{aligned} x_1 &: \{X_1\} \\ x_2 &: \{X_1, X_3\} \\ x_3 &: \{X_1, X_2\} \\ x_4 &: \{X_2, X_3\} \\ x_5 &: \{X_2\} \\ x_6 &: \{X_3\} \end{aligned}$$

An inverted index over a massive collection of sets with millions of tokens and millions of sets is very large, so the posting lists may have to be stored on disk or distributed as scalability becomes an issue. A query set may contain tokens not in the inverted index, and we need to determine this quickly. Thus, solutions typically store a data structure called a *dictionary*, that contains each token, its frequency and a pointer to its posting list.

Example 3.2.2. *The dictionary for our three set example (Example 3.2.1) and their index is given below.*

Token	Frequency	Posting List Pointer
x_1	1	1
x_2	2	2
x_3	2	3
x_4	2	4
x_5	1	5
x_6	1	6

The dictionary can be encoded as a hash table, a search tree, or an array based on a sorted order of tokens, for efficient look-up when processing query tokens [47].

Using the dictionary and inverted index there is a simple algorithm, that we call `MergeList`, for top-k overlap set similarity search [47]. First, initialize a hash map to store the intersecting token counts of *candidates* – sets that are discovered from the posting lists. For every query token, use the dictionary to check if the token exists in the inverted index; if it exists then read the entire posting list and increment counts for candidates in the list. Once all posting lists are read, we can sort the candidates by their intersecting token counts, and return the k sets with highest counts. The total time (ignoring memory hierarchy concerns) is

$$\sum_{x_i \in Q \cap U} L(f_i) \tag{3.1}$$

where U is the set of all tokens in the dictionary, f_i is the frequency of token x_i, and $L(\cdot)$ is the time to read a posting list as a function of the length of the list.

`MergeList`'s read time is linear to the number of matched tokens. This is not a big issue when sets have less than a few hundred tokens, however, in the context of joinable table search, the sets extracted from columns can easily have thousands or even millions of tokens, as shown in Table 3.1. Thus, a better approach could be to reduce the number of posting lists read.

3.2.2 Prefix Filter

Prefix filter, an idea proposed by Chaudhuri et al., to solve the threshold version of the set similarity search problem [13]. Xiao et al. used it to develop a solution for the top-k problem [69].

To explain prefix filter, let us first start with the threshold problem. Assuming an intersection size threshold, t, is given, and all candidates with intersection size greater than or equal to t must be returned. For simplicity assume all tokens in the query Q are found in the dictionary. We only need to read posting lists for a subset (any subset will do) of Q with size $|Q| - t + 1$. This is because for any candidate X such that $|Q \cap X| \geq t$,

$$|Q| - t + 1 > |Q| - |Q \cap X|$$

So even if all the tokens not in the intersection between Q and X (the count of which is the right-hand-side above) are selected as part of the subset, there will still be one token in the subset belonging to the intersection. A $|Q|-t+1$ subset of tokens is called a *prefix* of the query, and by reading the posting lists of a prefix, we are guaranteed to not miss any candidate that meets the threshold t, in another words, those sets pass the *filter* created by the prefix. However, for every candidate encountered in the posting lists read, we must compute their exact overlap to verify if they indeed meet the threshold.

So how do we use prefix filter in solving the top-k problem? The main idea is that given an intersection size threshold t, all candidates X such that $|Q \cap X| \geq t$ must be found in any subset (*prefix*) of posting lists given the subset's size is $|Q| - t + 1$, and posting lists outside the subset do not need to be read.

Xiao et al. proposed a top-k algorithm that uses prefix filter. It uses a fixed-size min-heap to keep track of the running top-k candidates [69]. The algorithm starts by reading the first posting list, and for every new candidate encountered, it computes the exact intersection size. If the intersection size is greater than the current k-th candidate's intersection size, the new candidate is pushed to the heap and the old k-th candidate is popped out. Otherwise the heap is not touched. The main trick of the algorithm is to use the running k-th candidate's intersection size as the threshold: after finishing reading a posting list, we need to check the overlap of the current k-th candidate X_k, $|Q \cap X_k|$, and if the number of lists already read so far is equal to $|Q| - |Q \cap X_k| + 1$, then the algorithm stops reading new lists and returns the current running top-k candidates as the result. Since $|Q \cap X_k|$ is the threshold, the first $|Q| - |Q \cap X_k| + 1$ posting lists become the prefix. We call this algorithm `ProbeSet`, because as opposed to `MergeList` that reads only posting lists, it probes candidates as it encounters them.

Compared to `MergeList`, `ProbeSet` offers the benefit of reading fewer posting lists, at the expense of probing some sets. However, the trade-off may be different in the context of searching joinable tables where sets may be very large, potentially making reading sets more expensive than reading the skipped posting lists.

3.2.3 Token Ordering and Position Filter

As described in the last section, `ProbeSet` reads and computes exact overlap for every new candidate encountered. Xiao et al. introduced an optimization techinique called *position filter* that prunes out candidates whose intersection sizes are less than a threshold before reading them [70]. For the rest of this paper, `ProbeSet` always uses the position filter.

There are two requirements to use position filter. First, we must assign a global ordering (e.g., lexicalgraphic, length, etc.) for the universe of all tokens, and all sets (including the query set) must be sorted by the global ordering. The choice of ordering does not affect the applicability of position filter. however, as we will show in later sections, the ordering can affect performance. The order assignment and sorting can be done during the indexing phase before building posting lists. The sorting of the query set can be done at the same time when matching with the dictionary.

Given all sets are sorted by a global ordering, the second requirement is that each token's posting list must contain the positions of the token in the sets that contain it, as well as the sizes of the sets. That is, instead of storing just the pointer to sets, each posting list also stores a position integer and a size integer associated with each set.

Example 3.2.3. *Using a similar example of sets and posting lists from Section 3.2.1, and assume the* *global ordering is indicated by a token's subscript, the posting lists become:*

$$\begin{aligned} X_1 &= \{x_1, x_{100}, x_{200}\} \\ X_2 &= \{x_2, x_5\} \\ X_3 &= \{x_2\} \\ X_4 &= \{x_2, ..., x_{100}, x_{101}\} \end{aligned} \implies \begin{aligned} x_1 &: \{(X_1, 1, 3)\} \\ x_2 &: \{(X_2, 1, 2), (X_3, 1, 1), \\ &\quad (X_4, 1, 100)\} \\ x_{100} &: \{(X_1, 2, 3), (X_4, 99, 100)\} \\ x_{200} &: \{(X_1, 3, 3)\} \end{aligned}$$

In each entry of a posting list, the second integer is the position, and the third integer is the size of the set.

So how can we apply the position filter to optimize `ProbeSet`? First, we have to sort the query set according to the global token ordering, and read the posting lists by the same ordering. When we encounter a new candidate X from the posting list of token x_i (i.e., the i-th token in the sorted query set), we can compute the upper-bound of its intersection size with the query set Q using the equation:

$$|Q \cap X| \leq |Q \cap X|_{ub} = 1 + \min\big(|Q| - i, |X| - j_{X,0}\big) \tag{3.2}$$

Where $j_{X,0}$ is the position of token x_i in X, it is also the first overlapping token between Q and X, hence the 0 in the subscript. Since both Q and X are sorted using the same global ordering and we are seeing X for the first time, we know for sure that there is no intersecting token between Q and X before this token at their i-th and $j_{X,0}$-th positions respectively. So the upper-bound intersection size is one plus the minimum of the number of remaining tokens in both sets. As we have discussed in Section 3.2.2, the threshold for top-k search problem is the running k-th candidate's intersection size, $|Q \cap X_k|$. Thus, if $|Q \cap X|_{ub} \leq |Q \cap X_k|$, we can skip X, and ignore it in future encounters.

Another benefit of using the position filter is the it reduces the amount of a candidate set that needs to be read. Since there is no intersecting token between Q and X before their first matching positions, we only need to read the suffix $X[j_{X,0} + 1 :]$ to compute the exact intersection size.

Example 3.2.4. *Consider Example 3.2.3's posting lists and let the query set be* $Q = \{x_1, x_2, x_{100}, x_{200}\}$ *and* $k = 2$. *We first read the posting list of* x_1*, which leads us to read and compute the exact intersection size for* X_1 *which is three. Then we read the posting list of* x_2*. Since the heap is not full, we read* X_2*, compute the exact intersection size, and push* $(X_2, 1)$ *to the heap. The running heap is* $\{(X_1, 3), (X_2, 1)\}$ *and the k-th candidate's intersection size is one.*

Now the heap is full and we have running top-k, we can use the position filter to check the next candidate X_3*. For* X_3*, because* $|Q \cap X_3|_{ub} = 1 + \min(4 - 2, 1 - 1) = 1 \leq 1$*, it does not pass the position filter. So we skip* X_3*. For* X_4*, because* $|Q \cap X_4|_{ub} = 1 + \min(4 - 2, 100 - 1) = 100 > 1$*, it passes the position filter. So we read* X_4*, and compute the exact intersection size which is 2. We pop and push the heap, which is now* $\{(X_1, 3), (X_4, 2)\}$.

We continue to read the third posting list x_{100}*, and we can skip both* X_1 *and* X_4 *as we have seen them before. The prefix size is* $4 - 2 + 1 = 3$*, thus* x_{100} *is the last posting list. We skip* x_{200} *and terminate the search.*

The total read time of the optimized `ProbeSet` is

$$\sum_{i=1}^{p^*} L(f_i) + \sum_{X \in W \setminus V} S(|X[j_{X,0}:]|) \tag{3.3}$$

The first summation term is the total time spent in reading posting lists of the prefix, where p^* is the final prefix length and $p^* = |Q| - |Q \cap X_k^*| + 1$, where X_k^* is the final k-th candidate in the final result. The second summation term is the total time spent in reading all qualified candidates ($W \setminus V$) encountered in the prefix, where W is the set of all candidates, and V is the set of candidates pruned by the position filter. The position $j_{X,0}$ is the first matching token position of X with Q. Lastly, $S(\cdot)$ is the time for reading a set as a function of its size.

3.3 A New Framework

`ProbeSet` takes advantage of the prefix filter to read fewer posting lists, but requires extra work, which is partially reduced by using position filter, to verify the candidates by reading and computing exact intersection sizes. In this section, we present a novel framework that quantifies the benefits (i.e., reduced cost) of reading posting lists and candidates, and an algorithm, called `JOSIE`, that uses this framework.

3.3.1 To Read or Not to Read

Example 3.3.1. *Let us revisit Ex. 3.2.4. Suppose after reading posting list x_2: instead of reading X_2 before X_3 and X_4, let us read X_4 first. The running heap after reading X_4 is $\{(X_1, 3), (X_4, 2)\}$, and by using the position filter, $|Q \cap X_2|_{ub} = 1 + \min(4-2, 2-1) = 2 \leq 2$, and $|Q \cap X_3|_{ub} = 1 + \min(4-2, 1-1) = 1 < 2$, we can skip both X_2 and X_3 and terminate.*

In this example, we read only 2 sets X_1 and X_4, compared to three sets in Example 3.2.4. The take away is that we do not have to read a candidate immediately after we counter it. By prioritizing reading some candidates before the others, we can "lift" the running k-th intersection size higher, increasing the pruning power of the position filter, and read fewer candidates.

Example 3.3.2. *Now suppose we make a different change after reading posting list x_2: instead of reading any sets, we continue to read posting list x_{100}.*

The posting list x_{100} does not give us any new candidates – both X_1 and X_4 are seen, however, the new position information allows us to update the position filters for the seen candidates: for X_4, its last token we saw before x_{100} was x_2, thus no intersecting tokens between x_2 (position 1) and x_{100} (position 99), and the only possible intersecting tokens exist in the remaining $100 - 99 = 1$ token after x_{100}. Using this new information, the position filter of X_4 becomes $|Q \cap X_4|_{ub} = 1 + 1 + \min(4-3, 100-99) = 3$, and most importantly, we only need to read the last token from X_4 (not the whole set) to compute its exact intersection size. The rest is the same as Example 3.3.1: we can safely ignore X_2 and X_3 using position filters, and terminate. The complete read sequence is shown in Figure 3.1.

In this example, we read in total three tokens from candidates, as shown in Figure 3.1: 2 from X_1 and 1 from X_4, while in Example 3.2.4 we read $2 + 1 + 99 = 102$ tokens and in Example 3.3.1 we read $2 + 99 = 101$ tokens. This example brings up two points: first, we do not need to read all candidates before reading the next posting list, as long as we come back to them (when necessary) before terminating

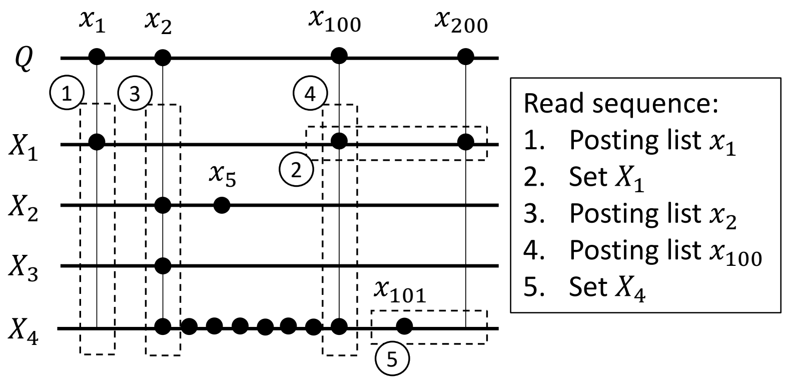


Figure 3.1: An example read sequence that alternates between reading posting lists and candidates. Each set is a horizontal line, and each posting list is a vertical line, which connects common tokens among sets.

search, we are still guaranteed to find the exact top-k; second, as Figure 3.1 shows, reading a posting list (e.g., x_{100}) has the benefit of potentially improving the position filters of unread candidates, in addition to reducing the amount of data (i.e., number of tokens) we have to read from candidates to compute exact intersection sizes.

As we can see from these two examples, reading a candidate and reading a posting list each has its own benefit in terms of reducing the time spent in reading other posting lists or candidates. These benefits have not been defined by the previous work and are data dependent. A quantitative comparison of these benefits leads to an approach that alternates between reading posting lists and reading sets, as illustrated by Figure 3.1. In the following sections, we will analyze these benefits quantitatively using a framework based on statistical approximation techniques, and present an adaptive algorithm for exact top-k overlap set similarity search that utilizes this framework.

3.3.2 Reading Candidates

As shown in Example 3.3.1, one potential benefit of reading a candidate is to increase the running k-th intersection size (i.e., increase $|Q \cap X_k|$) which is used by both the prefix filter and position filter. Recall, the prefix filter prunes out posting lists (those beyond the prefix) and all *unseen* candidates in these lists. On the other hand, the position filter prunes out *seen* candidates without reading them, by comparing the upper-bound intersection size of a candidate with threshold $|Q \cap X_k|$.

Now the problem is how do we know whether a candidate can make it into the running top-k and increase the threshold $|Q \cap X_k|$? Let X be the current candidate. Should we read it? We do not have full information about all the tokens in X, however, we actually know quite a lot:

- $i_{X,0}$ is the position in Q of the first token that intersects with X, where X first appears in the posting list of that token;
- $j_{X,0}$ is the position in X of the first token that intersects with the query;
- j_X is the position in X of the most recent token that intersects with the query; and
- $|Q[1\!:\!i] \cap X|$ is the number of intersecting tokens between Q and X up to and including the token at the current position i in Q, where $Q[1\!:\!i]$ is the prefix of Q up and including token x_i: $x_1, x_2, ..., x_i$.

Both j_X and $|X|$ can be found in the posting list entry of X. We read posting lists of Q's tokens from left to right according to the global token order, and keep track of $i_{X,0}$, j_X, $|Q[1\colon i] \cap X|$ and $|X|$ for all seen candidates. As explained in the previous section, we do not have to read every candidate as we encounter it, as long as we read it or prune it before terminating. Figure 3.2 illustrates.

Given this information about X, we can estimate its intersection $|Q \cap X|$, which, together with the running top-k heap, can be used to approximate the new threshold $|Q \cap X_k|$.

If we consider the set $Q \cap X$ as a subset of $Q[i_{X,0}\colon]$ – the subset of Q starting from the first matching token with X, and assume the members of $Q \cap X$ are uniformly distributed over $Q[i_{X,0}\colon]$, an unbiased estimator for $|Q \cap X|$ is

$$\begin{aligned} |Q \cap X|_{est} &= \frac{|Q[i_{X,0}\colon i] \cap X|}{i - i_{X,0} + 1} \times (|Q| - i_{X,0} + 1) \\ &= \frac{|Q[1\colon i] \cap X| - \cancel{|Q[1\colon i_{X,0}] \cap X|}}{i - i_{X,0} + 1} \times (|Q| - i_{X,0} + 1) \\ &= \frac{|Q[1\colon i] \cap X|}{i - i_{X,0} + 1} \times (|Q| - i_{X,0} + 1) \end{aligned} \tag{3.4}$$

if we consider $Q[i_{X,0}\colon i]$ as a random sample of $Q[i_{X,0}\colon]$. A proof of this can be done using techniques introduced in Broder [9]. If the sample $Q[i_{X,0}\colon i]$ is small, the variance will be high. Therefore, we should read a few posting lists starting from position $i_{X,0}$ before using this estimation.

Armed with the knowledge of $|Q \cap X|_{est}$, we can estimate the new threshold $|Q \cap X'_k|$, where X'_k is the new k-th candidate after reading X. We first compare $|Q \cap X|_{est}$ with the current threshold $|Q \cap X_k|$. If $|Q \cap X|_{est} \leq |Q \cap X_k|$, then we assume the candidate X likely will not qualify for the running top-k, and the current threshold is unchanged. If $|Q \cap X|_{est} > |Q \cap X_k|$, we then look at what the new threshold would be after pushing X to the heap (and popping X_k), by comparing $|Q \cap X|_{est}$ with $|Q \cap X_{k-1}|$, where X_{k-1} is the $(k-1)$-th candidate (done in constant time with a binary heap). The estimation can be summarized using Equation 3.5 below.

$$|Q \cap X'_k|_{est} = \begin{cases} |Q \cap X_k| & \text{if } |Q \cap X|_{est} \leq |Q \cap X_k| \\ |Q \cap X|_{est} & \text{if } |Q \cap X|_{est} > |Q \cap X_k| \\ & \text{and} |Q \cap X|_{est} \leq |Q \cap X_{k-1}| \\ |Q \cap X_{k-1}| & \text{if } |Q \cap X|_{est} > |Q \cap X_{k-1}| \end{cases} \tag{3.5}$$

Now using the new threshold, we can finally estimate the benefit of reading candidate X in terms of time saved. As mentioned earlier, the benefit consists of two parts. The first part is from eliminated posting lists through the prefix filter update. Let the current prefix length be $p = |Q| - |Q \cap X_k| + 1$, and the new prefix length after reading X be $p' = |Q| - |Q \cap X'_k|_{est} + 1$. Then the posting lists from $p' + 1$ to p inclusive will be eliminated, and the benefit is equal to the sum of the time to read these posting lists.

Another benefit is from eliminating candidates by updating the position filter. The position filter for a candidate Y has upper-bound intersection size:

$$|Q \cap Y|_{ub} = |Q[1\colon i] \cap Y| + \min(|Q| - i, |Y| - j_Y) \tag{3.6}$$

This is different from Equation (3.2): the first term on the right is the number of intersecting tokens

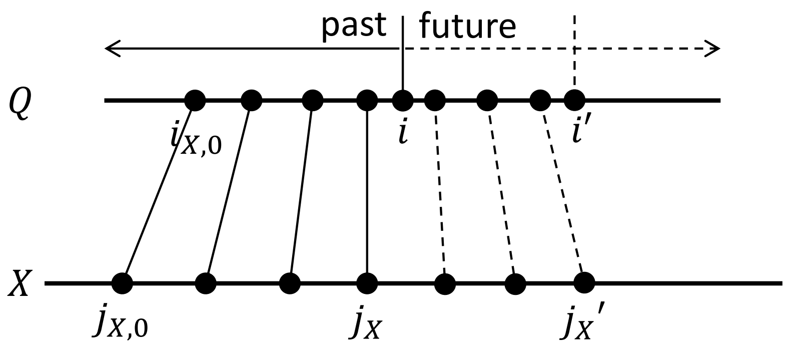


Figure 3.2: The query set Q and candidate X after reading the posting list at position i. Pairs of dots with solid lines represent intersecting tokens, pairs with dotted lines represent future intersecting tokens, and non-intersecting tokens in both sets are not shown.

seen so far, as we do not read Y immediately after encountering it, this number may no longer be one. If $|Q \cap Y| \leq |Q \cap X'_k|_{est}$, then candidate Y will likely be eliminated after reading X. Thus, using the updated position filter, we can determine whether a candidate will be eliminated, and the benefit is the sum of the time it would take to read the eliminated candidates.

Equation 3.7 gives the benefit of reading candidate X. The set W_i is all unread candidates at posting list i and $I(\cdot)$ is an indicator function that evaluates to one only if the condition argument is true, and zero otherwise.

$$\begin{split}Benefit(X) = \sum_{i=p'+1}^{p} L(f_i) + \\ \sum_{Y \in W_i, Y \neq X} S(|Y[j_Y + 1:]|) \cdot I(|Q \cap Y|_{ub} \leq |Q \cap X'_k|_{est})\end{split} \tag{3.7}$$

3.3.3 Reading Posting Lists

We now provide a quantitative framework for estimating the benefit of reading posting lists.

As before, let i be the position of the current posting list that we just read. As discussed in the previous section, in order to avoid a large variance in estimating intersection size, we need to initially read a few posting lists for each candidate. So posting lists are read in "batch", and we use i' to indicate the position of the end, or the last posting list, of the next batch. This is shown in Figure 3.2, which also shows the intersecting tokens between query Q and a candidate X. Let j_X be the most recent position in X whose token intersects with Q, and j'_X be the position in the future after reading the next batch of posting lists ending at i'.

Reading the next batch always improves (or at worse leaves unchanged) the pruning power of the position filter, by tightening the upper-bound for all candidates. This can be understood using the future upper-bound intersection size for candidate X at i':

$$\begin{aligned}
|Q \cap X|'_{ub} &= |Q[1\!:\!i'] \cap X| + \min(|Q| - i', |X| - j'_X) \\
&= |Q[1\!:\!i] \cap X| + |Q[i+1\!:\!i'] \cap X| \\
&\quad + \min(|Q| - i', |X| - j'_X) \\
&\le |Q[1\!:\!i] \cap X| + |Q[i+1\!:\!i']| \\
&\quad + \min(|Q| - i', |X| - j'_X) \\
&\le |Q[1\!:\!i] \cap X| + \min(|Q| - i, |X| - j_X) \\
&= |Q \cap X|_{ub}
\end{aligned} \tag{3.8}$$

Intuitively, reading the next batch verifies the exact intersection in that batch, and "exposes" the number of tokens that do not intersect but have been consider in the upper-bound. So the total number of tokens we can count toward the upper-bound is reduced, and the upper-bound is lowered.

Given the new upper-bound $|Q \cap X|'_{ub}$, by comparing it with the current k-th candidate's intersection size, or the current threshold, $|Q \cap X_k|$, we know for every candidate X whether it would be eliminated or not. If not, we may still have some benefit because we know we would only need to read the tokens in $X[j'_X + 1\!:]$.

So now the question is how do we estimate $|Q \cap X|'_{ub}$. Calculating $|Q \cap X|'_{ub}$ requires $|Q[i+1\!:\!i'] \cap X|$, the number of intersecting tokens between Q and X in the next batch, and j'_X, the future position of the last intersecting token.

First, we can estimate $|Q[i+1\!:\!i'] \cap X|$ using the same method we used for the total intersection size, $|Q \cap X|$, in Equation 3.4, by assuming uniform distribution of intersecting tokens over the range $Q[i_{X,0}\!:]$.

$$|Q[i+1\!:\!i'] \cap X|_{est} = \frac{|Q[i_{X,0}\!:\!i] \cap X|}{|Q| - i_{X,0} + 1} \cdot (i' - i) \tag{3.9}$$

Since we keep track of $i_{X,0}$ and $|Q[i_{X,0}\!:\!i] \cap X|$ for every candidate X, we can compute this estimate.

Second, we can estimate j'_X by first leveraging the fact that the number of intersecting tokens between $i+1$ and i' in Q must be equal to that between $j_X + 1$ and j'_X in X, as shown in Figure 3.2. Then, we can apply the same estimation method for intersection size in Equation 3.9 to create an approximate equality, from which we derive an expression for $j'_{X,est}$.

$$\begin{gathered}
|Q[i+1\!:\!i'] \cap X|_{est} \approx |Q \cap X[j_X + 1\!:\!j'_X]|_{est} \\
\frac{\cancel{|Q[i_{X,0}\!:\!i] \cap X|}}{|Q| - i_{X,0} + 1} \cdot (i' - i) \approx \frac{\cancel{|Q \cap X[j_{X,0}\!:\!j_X]|}}{|X| - j_{X,0} + 1} \cdot (j'_X - j_X) \\
j'_{X,est} = j_X + \frac{i' - i}{|Q| - i_{X,0} + 1} \cdot (|X| - j_{X,0} + 1)
\end{gathered} \tag{3.10}$$

In the above derivation, since $|Q[i_{X,0}\!:\!i] \cap X|$ is exactly the same as $|Q \cap X[j_{X,0}\!:\!j_X]|$ (see Figure 3.2), we can cancel them out on both sides of the equation.

Substituting $|Q[i+1\!:\!i'] \cap X|_{est}$ and $j'_{X,est}$ for $|Q[i+1\!:\!i'] \cap X|$ and j'_X, respectively, in Equation 3.8, we can obtain the estimation for the future upper-bound of X in the position filter.

$$\begin{aligned} |Q \cap X|'_{ub,est} = |Q[1\colon i] \cap X| + |Q[i+1\colon i'] \cap X|_{est} \\ + \min(|Q| - i', |X| - j'_{X,est}) \end{aligned} \tag{3.11}$$

Equipped with the estimation for the upper-bound, we can finally summarize the benefit of reading the next batch of posting lists for tokens $B_{i+1,i'} = x_{i+1}, ..., x_{i'}$.

$$\begin{aligned} & Benefit(B_{i+1,i'}) \\ & = \sum_{X \in W_i} S(|X[j_X\colon]|) \cdot I(|Q \cap X|'_{ub,est} \leq |Q \cap X_k|) \\ & + \big(S(|X[j_X\colon]|) - S(|X[j'_X\colon]|)\big) \cdot I(|Q \cap X|'_{ub,est} > |Q \cap X_k|) \end{aligned} \tag{3.12}$$

The first term is the time spent on reading candidate X, corresponding to the cost saved due to the elimination of X by reading the next batch of posting lists. The second term is the reduction in read time for reading X when it is not eliminated, but due to an updated position filter, fewer of its tokens need to be read. As in Equation 3.7, $I(\cdot)$ is an indicator function. It is important to note that, the benefit of reading posting lists is always non-negative.

3.3.4 An Adaptive Algorithm

Given our quantitative framework for estimating the respective benefits of reading a candidate and reading a batch of posting lists (in terms of the amount of read time saved), we now calculate the *net cost* as the difference between the read time incurred and the read time saved.

$$\begin{aligned} NetCost(X) = S(|X[j_X+1\colon]|) - Benefit(X) \\ NetCost(B_{i+1,i'}) = \sum_{x=i+1}^{i'} L(f_x) - Benefit(B_{i+1,i'}) \end{aligned} \tag{3.13}$$

Clearly, the lower the net cost, the better the performance.

We now present our algorithm, `JOSIE` (**JO**ining **S**earch using **I**ntersection **E**stimation), that prioritizes reading posting lists or candidates based on net cost computed using estimated intersection sizes. The pseudo code is shown in Algorithm 2.

Starting from reading the first batch of posting lists, the algorithm relies on the functions NetCost(X) and NetCost($B_{i+1,i'}$) to determine whether to read the best unread candidate, ranked by net cost, or to read the next batch of posting lists. Each time a candidate is read, the algorithm updates a running top-k heap and the prefix filter. Each time posting lists are read, the algorithm updates the pointers. Either way, after the read, the algorithm applies the position filter and eliminates unqualified candidates.

Since the threshold $|Q \cap X_k|$ is only valid (or non-zero) after at least k candidates are read, `JOSIE` always chooses to read candidates before the top-k heap is full ($|h| = k$), unless it is reading the first batch ($u = \emptyset$ and $X = \emptyset$).

The algorithm terminates when the posting list pointer i is outside of the prefix filter range ($i > p$), and there is no unread candidate ($u = \emptyset$). If $i > p$ but $u \neq \emptyset$, it means that the algorithm still has to finish the remaining candidates by either reading posting lists or reading sets.

Algorithm 2 Algorithm `JOSIE` using the cost model

```
procedure JOSIE($U, I, Q, k, \lambda$)  ▷ $U, I$ are the dictionary and inverted index, and $\lambda$ is the batch size
    $Q \leftarrow Q \cap U$  ▷ Use the dictionary
    $x_1, x_2, ..., x_n \leftarrow$ Sort($Q$)  ▷ Apply global ordering
    $h \leftarrow \{\}$  ▷ $h$ is the heap for running top-k sets
    $u \leftarrow \{\}$  ▷ $u$ is the hash map of unread candidates
    $i \leftarrow 1$, $i' \leftarrow 1 + \lambda$, $X_k \leftarrow \emptyset$  ▷ $X_k$ is the $k$-th candidate
    $p \leftarrow |Q| - |Q \cap X_k| + 1$
    while $i \leq p$ or $u \neq \emptyset$ do
        $X \leftarrow$ Best($u$)
        if $|h| = k$ and NetCost($X$) > NetCost($B_{i+1,i'}$) or ($u = \emptyset$ and $X = \emptyset$) then
            $u \leftarrow u \cup$ Read($I, B_{i+1,i'}$)  ▷ Read posting lists
            $i \leftarrow i'$, $i' \leftarrow i' + \lambda$
        else
            TryPopPush($h, k, X, |Q \cap X|$)  ▷ Read set $X$
            $X_k, |Q \cap X_k| \leftarrow$ Head($h$)
            $p \leftarrow |Q| - |Q \cap X_k| + 1$
        end if
        $u \leftarrow$ PositionFilter($u$)  ▷ Eliminate candidates
    end while
    return PopAll($h$)
end procedure
```

The total time of `JOSIE` is

$$\sum_{i=1}^{p^*} L(f_i) + \sum_{i=p^*+1}^{p^*+\delta} L(f_i) + \sum_{X \in W \setminus V^*} S(|X[j_X :]|) \tag{3.14}$$

The formula is similar to `ProbeSet`'s (Equation 3.3) however it has a few differences. First, in `ProbeSet`, no more posting lists are read after the last position in the final prefix, indicated by p^*, while in our algorithm, the net cost of reading the next batch of posting lists may be less than reading the next candidate, leading to more posting lists read. We use δ to indicate the extra posting lists read after the final prefix position. It is important to note that the extra posting lists are only used to help reduce the read time of existing candidates, and do not introduce any new candidate, as those candidates will be pruned automatically by the prefix and position filters.

The second difference is the last term. Instead of reading every qualified candidate as we encounter it as in `ProbeSet`, we read candidates in increasing order of their net costs, and thus the candidates with the highest pruning effect (i.e., benefit) get read first. Thus, we are always going to read no more candidates than `ProbeSet` and often many fewer. That is, $V \subseteq V^*$, where V is the set of candidates that are pruned at first encounter only, and V^* is the set of all candidates that are pruned in our algorithm. Note that $V^* = \emptyset$ is extremely unlikely as it happens only when the candidates appear in the order of strictly increasing intersection sizes with the query.

The last difference is the read time for each candidate. `ProbeSet` reads a qualified candidate at first encounter, thus the read time is $S(|X[j_{X,0} :]|)$. Our algorithm, on the other hand, reads a candidate at some posting list after the first encounter, due to reading in batch and using the cost model. Thus, we will almost always have fewer tokens to read for any candidate (in the worst case read the full set like `ProbeSet`). Thus less time is spent on reading the candidates.

In conclusion, our algorithm reads fewer candidates than `ProbeSet`, and often a smaller portion of those candidates. This reduction in read time is at the expense of reading the extra δ number of posting lists. However, due to the use of cost model, every step of the algorithm chooses the direction with the least net cost, thus the total cost of reading the extra posting lists is likely much less than the reduced cost of reading candidates.

3.4 A Practical Search Engine

We now present several optimizations that improve the performance of JOSIE, and practical considerations for estimating read cost efficiently.

3.4.1 Distinct Posting Lists

Although the cost model works with any global order, for this work, we use increasing frequency order as the global order, because it tends to minimize the number of candidates in the prefix, as suggested by Chaudhuri et al. [13]. Another advantage of using frequency order is that duplicate posting lists are together, and we use this to further optimize the search engine.

Two posting lists are duplicates if they point to the same sets. Because of the frequency ordering (and within a frequency we order lists by some fixed ordering on sets), duplicate lists will be adjacent. Importantly, we observed many duplicate posting lists in both Open Data and Web Tables. The statistics are in Table 3.2.

Table 3.2: Posting lists in Open Data and Web Tables.

	#Original Tokens	% Tokens with Dup	# Tokens After De-dup
Open Data	563,320,456	98%	9,003,658
Web Tables	184,644,583	83%	45,395,793
Enterprise Data	3,902,604	99.87%	9,133

Even though reading multiple duplicate posting lists does not yield more candidates than reading just one, it still provides the benefit of reducing the cost of reading existing candidates, as discussed in Section 3.3.3. So how can we avoid reading duplicate lists, while still getting the benefit?

To avoid reading duplicate posting lists, we assign each token in the dictionary a *duplicate group ID*, which is a unique identifier for a group of duplicate posting lists. So when matching a query with the dictionary, a duplicate group ID is also mapped to every query token, in addition to the frequency and posting list pointer.

The cost model assumes the posting lists of a query set are read sequentially in the frequency order of their tokens. When going through the posting lists, we read just the posting list of the last token in each group present in the query. This can be done by checking if the next token has the same or different duplicate group ID as the current one. If the next one has the same duplicate group ID, then the posting list can be skipped. This idea can be illustrated using the example in Figure 3.3: the first and second posting lists with duplicate group ID 1 are skipped, and the first two posting lists with duplicate group ID 5 are also skipped.

We modified our cost model to handle duplicates. First, we count groups, rather than lists when forming batches and ignore the skipped posting list in calculating the read cost. Second, we need to

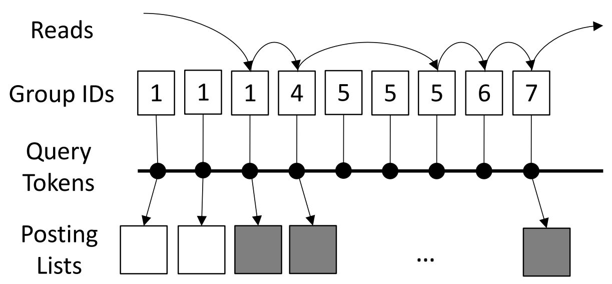


Figure 3.3: A read sequence of distinct posting lists.

account for the number of posting lists skipped when reading the last posting list in the same duplicate group. This ensures that the information about candidates (see Section 3.3.2), such as the number of intersecting tokens see so far, is updated correctly.

3.4.2 Indexing Data Lakes

In this section, we discuss how we build an inverted index from a data lake in three major steps, implementation details, and a strategy for incremental updates.

Extracting Raw Tokens

In the first step, we extract sets from tables in the data lake. Each set is assigned a `SetID`, a unique integer identifier for the original table and column from which the set is extracted. Each column (set) value is a token. Sets are flattened into a mapping called `RawTokens` of tuples `(RawToken, SetID)`.

Building Token Table

In the second step, we build a mapping called `TokenTable`, that maps every token `RawToken` to a unique integer ID, `TokenID`, which indicates the token's position in the global order, and an integer duplicate group ID, `GroupID`. Each tuple of the mapping is `(RawToken, TokenID, GroupID)`.

To build a `TokenTable`, we first build posting lists of sorted `SetID`, by grouping tuples in `RawTokens` by `RawToken`. Then we sort all posting lists by length (i.e., token frequency), MurmurHash3 hash values[2] of the posting lists, and then the posting lists themselves. The use of MurmurHash3 hash helps us to avoid most of the expensive pair-wise comparisons of same-length posting lists that did not have hash collision. Then we use the sorted position of each posting list as the `TokenID` of the corresponding `RawToken` to obtain the mapping `(RawToken, TokenID)`. It is important to note that, even though we are using token frequency as the global order, our algorithm described in Section 3.3.4 works for any other global order, for example, the original positions of posting lists before sorting. In order to leverage the distinct list optimization described in Section 3.4.1, however, it is convenient to use frequency order.

As described in Section 3.4.1, each duplicate group spans a consecutive interval in the global order by frequency. To create duplicate groups, we must identify their starting and ending positions. The starting positions are identified by comparing every pair of adjacent posting lists (lower and upper) in the global order: for a pair, if the lower and upper posting lists are different, then the upper posting list's `TokenID` is the starting position of a new duplicate group. The first duplicate group's starting position

[2]https://github.com/aappleby/smhasher

Table 3.3: Indexing benchmark sets using Apache Spark on a cluster of 3 worker nodes each with 100 GB of memory and 63 cores.

	Open Data	Web Tables
Input `RawTokens`	37.3 GB	51.4 GB
Output Posting Lists	51.3 GB	36.0 GB
Output Integer Sets	10.6 GB	16.3 GB
Duration	2.4 h	1.1 h
Max Per-Task Peak Memory	7.1 GB	9.1 GB

is 0. Similary, the ending positions are identified by scanning every pair of adjacent groups' starting positions: the upper group's starting position is the ending position of the lower duplicate group. Once all the starting and ending positions are identified, we can generate the `GroupID` as the groups' positions in the sorted order, and the mapping `(TokenID, GroupID)` by enumerating from the starting to the ending position of every group.

Lastly in this step, we join the two mappings, `(RawToken, TokenID)` and `(TokenID, GroupID)` to obtain `TokenTable`.

Creating Integer Sets

The last step is to convert all tokens in sets to integers (`TokenID`). We use integers because computing intersection between integer sets is faster than between string sets, and storage systems such as Postgres[3] can read integer sets efficiently. Another benefit of using integer sets is better estimation of read cost, which is discussed in Section 3.4.3.

We create integer sets by first joining mappings `RawTokens` and `TokenTable` on `RawToken`, and then grouping by `SetID` to get sets while selecting only `TokenID` and `GroupID`. Each set is sorted by `TokenID` to reflect the global order. Lastly, we create the posting lists as shown in Section 3.2.3 from the integer sets.

Implementation and Performance

We implemented the indexing algorithm using Apache Spark[4]. The input is the `RawTokens` table of tuples `(RawToken, SetID)`, and the outputs are the integer sets and posting lists. Apache Spark automatically creates a series of tasks, some with inter-dependencies, for parallel execution on a cluster. Since Spark tasks can be scheduled one after another or simultaneously, the maximum per-task peak memory usage indicates the minimum amount of memory that must be available on each worker node before running into out-of-memory error. Table 3.3 lists the results on input and output sizes, duration, and maximum per-task peak memory.

Handling Incremental Updates

The posting lists and integer sets in our index can be updated incrementally. To do so we must maintain the global order of tokens to ensure the correctness of our algorithm as described in Section 3.3.4.

[3]https://www.postgresql.org/docs/10/static/intarray.html
[4]https://spark.apache.org

Any update to the index, such as adding a new set, a new token, or removing a token, can be expressed as a sequence of `ADD` and `DELETE` operations given `(RawToken, SetID)` tuples. For `ADD`, there are 4 different cases:

Case 1: `RawToken` and `SetID` both exist. This requires no action because a set consists of unique tokens.

Case 2: `RawToken` exists, and `SetID` is new. We create a new integer set using the existing `TokenID`. and then append to the posting list of `TokenID` a new entry $(\texttt{SetID}, 0, 1)$. See Section 3.2.3 for posting list entry. Due to the append, we invalidate the duplicate group ID of this posting list. The global order is unchanged.

Case 3: `RawToken` is new, and `SetID` exists. Because the token is not in our index, we assign a new integer `TokenID` by incrementing the maximum existing `TokenID`, effectively expending the global order by one and keeping the existing positions unchanged. Because the new token is at the end of the global order, we append its new `TokenID` to the end of the existing set given by the `SetID`. For existing posting lists of the set, we update the set's entry by incrementing the size: $(\texttt{SetID}, *, size + 1)$ – because the new token is appended, the existing tokens' positions are unchanged. Lastly, we create a new posting list for the new token with entry $(\texttt{SetID}, size, size + 1)$.

Case 4: `RawToken` and `SetID` are both new. This is the combination of the previous two cases. We first assign a new `TokenID` to the `RawToken`, then create a new integer set, and lastly create a new posting list with entry $(\texttt{SetID}, 0, 1)$. The global order of tokens is expanded by one, but the rest is unchanged.

This incremental update strategy can also be applied directly to an empty index and the resulting global order would be the order in which tokens are added.

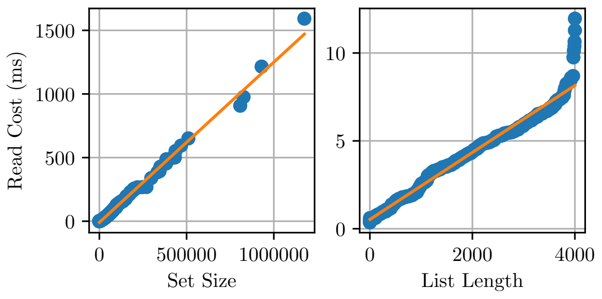
For `DELETE`, we first remove the entry from the posting list corresponding to `RawToken`, and update the entries in other posting lists of the tokens in the set: some positions need to be shifted by one, and the sizes are decremented by one. Then we also remove the token from the integer set.

In summary, the incremental update strategy keeps the global order of existing tokens unchanged, ensuring the correctness of our search algorithm (Section 3.3.4). The caveat is that we cannot assign duplicate group IDs for the newly added posting lists as well as those with append or removal, because finding the duplicate groups requires sorting all posting lists. Thus, we cannot skip the new and updated posting lists even if they are duplicates. This only affects the query performance, and since the tables in data lakes are often used for analytic tasks, updates are rare. The index can keep track of the number of posting lists affected and statistics on query runtimes, so that it can inform the system administrator to choose an appropriate time to rebuild the index for better performance.

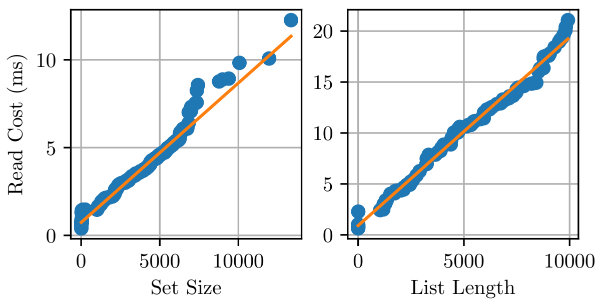
3.4.3 Estimating Costs

Throughout Section 3.3, we assume the cost of reading a candidate and a posting list can be calculated using functions $S(\cdot)$ and $L(\cdot)$ respectively. What function is suitable here?

There are different components involved when a candidate or a posting list is read. First, there is *index access time* – the time to find where the set or posting list is located in the storage layer (e.g., disk, memory, etc.) given its pointer. For Open Data, since there are more than 563M posting lists, this time can be significant. Then, there is the time to read the set or posting list from the storage layer, and the time to transfer the data through network or inter-process communication, which also involves serialization and deserialization. These components can be collectively called *read time*.



(a) Samples of read costs of sets (left) and posting lists (right) in Open Data and fitted lines.



(b) Samples of read costs of sets (left) and posting lists (right) in Web Tables (bottom), and fitted lines.

Figure 3.4: Estimating read costs of sets and posting lists using samples

The index access time is likely a constant, especially with popular indexes such as B+ Trees used by many storage layers. On the other hand, the read time is proportional to the size of data. When the storage system is disk-based, read may be accelerated through sequential I/O, however, the data still needs to be transferred to the search engine process, and that takes time proportional to the size of the data.

Thus, we use linear functions to express the read cost:

$$S(|X|) = s_0 + s_1 \cdot |X|,\ \ L(f_i) = l_0 + l_1 \cdot f_i \tag{3.15}$$

Where s_0 and l_0 are the index access times, and s_1 and l_1 are the factors for the read times.

In order to estimate the parameters in Equation 3.15, we sampled 1,000 sets and 1,000 posting lists and measure the I/O times, and then fit the functions to the sampled data points. Our experiments will use the same two data lakes from Chapter 2, namely Open Data and Web Tables. Figure 3.4a and 3.4b show the samples and fitted lines from sets and posting lists in Open Data and Web Tables, respectively.

3.5 Experiments

We now demonstrate the performance of JOSIE through experiments using real-world data lakes, and compare it to the state-of-the-art in both exact and approximate approaches.

3.5.1 Data Lakes

We used two data lakes in our experiments, Open Data (see Section 2.5.1) and Web Tables, (a public corpus [40]). We also removed all numerical values, as they create casual joins that are not meaningful. The characteristics of the extracted sets are shown in Table 3.1. Web Tables has 219× more sets than Open Data, while its average set size is much smaller (10 vs. 1,540). This means reading a set from the Web Table lake is usually cheaper. Tokens in Web Tables appear in more sets overall than those in Open Data. This implies that the posting lists of Web Tables tokens are often longer, and thus more expensive to read.

3.5.2 Query Benchmarks

We generated three query benchmarks from each data lake. Each benchmark is a set of queries (sets) selected from a size range in order to evaluate performance on different query sizes.

For Open Data, we used the same query benchmarks as described in Section 2.5.2: three benchmarks from ranges: 10 to 1k, 10 to 10k, and 10 to 100k.

For Web Tables, we used different ranges for benchmarks: 10 to 100, 10 to 1k, and 10 to 5k. The last range only has 5 intervals. This is because the sets in Web Tables tend to be much smaller than Open Data, and there are not enough sets in the 5k to 10k range to sample 100 sets for every interval. In order to make up for the total, we sampled 200 sets for each of the 5 intervals.

For a query set Q, and dictionary of all tokens U in the index excluding the query, we used $|Q \cap U|$ instead of $|Q|$ to decide which range the query belongs to. This is because any token that only exists in Q, but not in U will not have a posting list, and add uncontrolled noise to the experiment when we want to measure the effect of the number of posting lists on performance.

We have made the benchmarks available online[5].

3.5.3 Setup

We build an inverted index and dictionaries for each of the Open Data and Web Tables repositories using Apache Spark[6]. The posting lists and sets are then stored in a PostgreSQL[7] database as two separate tables, with B+ Tree indexes built on tokens and sets. All experiments are conducted on a machine with two Intel® Xeon® CPU E5-2620 v4 @ 2.10GHz (16 cores), 128 GB DDR4 memory, and an Intel® SSD DC S3520 3D MLC.

3.5.4 Algorithms

In our experiments, we will compare the following five algorithms.

`MergeList-D` This is based on the `MergeList` algorithm introduced in Section 3.2.1, modified to read only the distinct posting lists using the technique introduced in Section 3.4.1.

`ProbeSet-D` This is based on the `ProbeSet` algorithm introduced in Section 3.2.3, modified to also read only distinct posting lists. This algorithm relies on prefix filter to limit the number of posting lists to read, and uses position filter to reduce the number of candidates to read.

`LSHEnsemble-60` This is LSH Ensemble (from Chapter 2) using 32 partitions. $\frac{|Q \cap X|}{|Q|} \rightarrow R[0.0, 1.0]$, The original algorithm only supports threshold-based search, so we modified the algorithm to support top-k search by trying a sequence of decreasing thresholds starting at 1.0 with a step size equal to 0.05, until enough candidates have been acquired. Since this is an approximate algorithm, it can be extremely fast by just returning any candidate. So for a fair comparison we require the algorithm to keep acquiring and reading (distinct) candidates until a minimum recall of 60% is reached for the top-k candidates. The ground truth is provided to the algorithm for every query.

`LSHEnsemble-90` This is the same algorithm as the previous one, with the only difference being the minimum recall is 90%. A higher minimum recall means the algorithm has to read more candidates in order to improve accuracy.

[5]`https://github.com/ekzhu/set-similarity-search-benchmarks`
[6]`https://spark.apache.org`
[7]`https://www.postgresql.org`

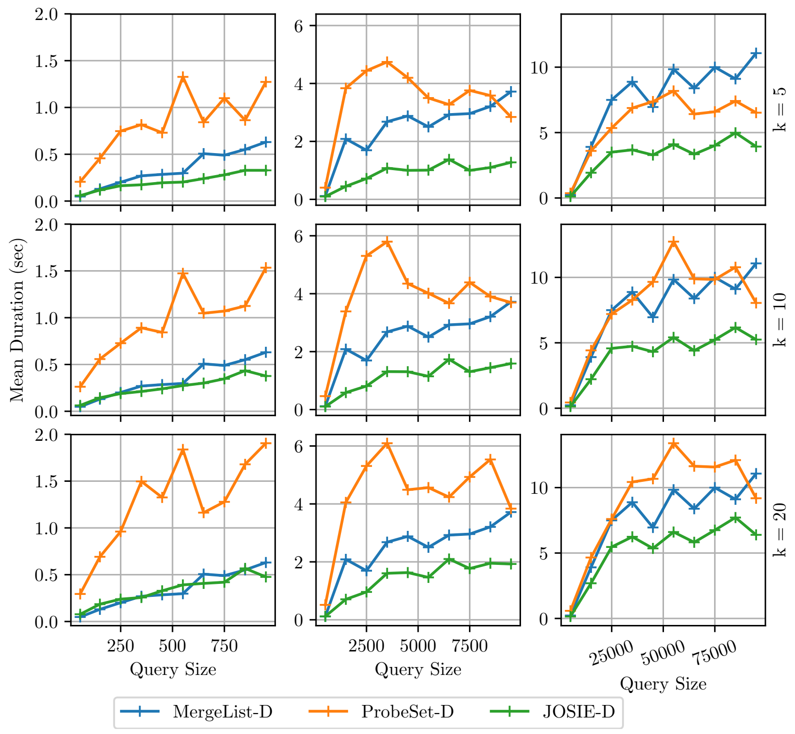


Figure 3.5: Mean query time, Open Data benchmark.

`JOSIE-D` This is the cost-based algorithm presented in Section 3.3.4, with the distinct posting list optimization.

3.5.5 Open Data Results

We compare the running time of `JOSIE-D` with the baseline algorithms `MergeList-D` and `ProbeSet-D` on benchmarks generated from Open Data tables. The result is shown in Figure 3.5. Each row corresponds to a different k, which is the number of top results to retrieve, and each column corresponds to a different query benchmark (i.e., 1k, 10k, and 100k). Each point is the mean running time of benchmark queries from a single interval (i.e., 100 queries).

`ProbeSet-D`'s performance deteriorates quickly as k increases. This is because with larger k, the prefix filter is larger and has less pruning power, as the k-th candidate's intersection size can be much lower. Thus, `ProbeSet-D` must read more posting lists and all the candidates appear in those posting lists. In contrast, `MergeList-D` simply reads all the distinct posting lists and none of the candidates, regardless of k. So its performance is constant with respect to k.

We can also observe that `ProbeSet-D`'s running time is much more volatile than `MergeList-D`, as shown in the first row of Figure 3.6, which plots the standard deviation of query duration. This is because `ProbeSet-D` reads all candidates it encounters and has no control over the sizes of candidates it reads, thus its running time is heavily dependent on the distribution of posting list lengths (i.e., token frequencies) and set sizes.

Now we look at our proposed algorithm `JOSIE-D`. It out-performs both `ProbeSet-D` and `MergeList-D`, on all benchmarks and all ks, often by 2 to 4 times, except for the $k = 20$ and 1k benchmark, on which it is on par with `MergeList-D`.

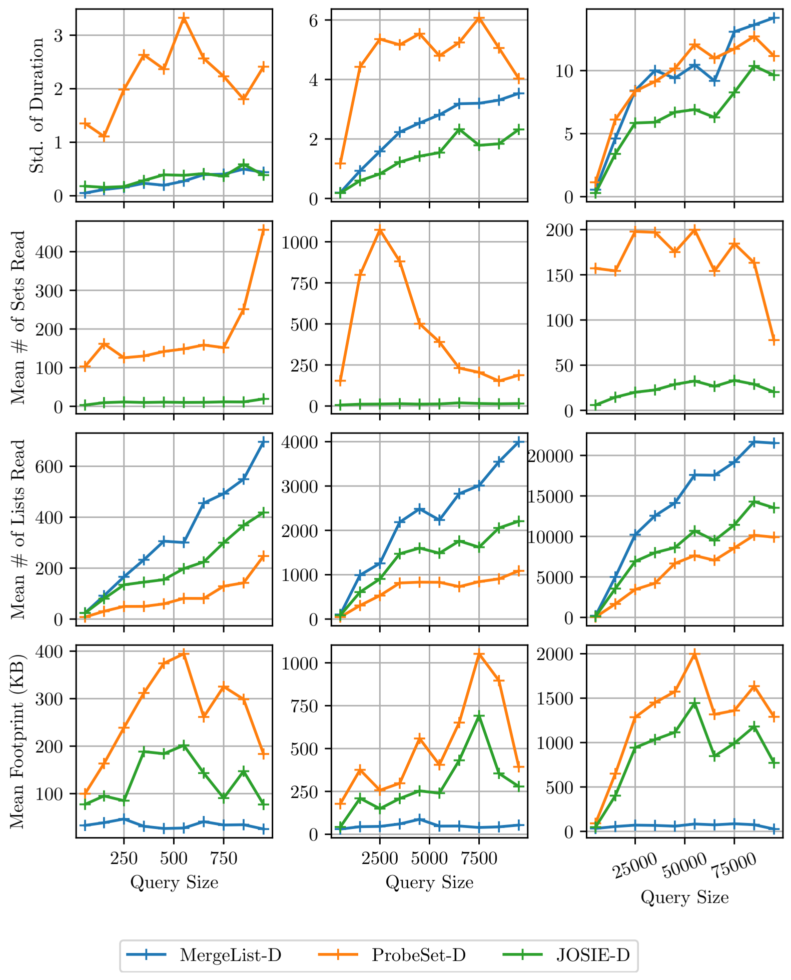


Figure 3.6: Standard deviation of durations, mean number of set and posting lists read per query, and query memory footprint on Open Data benchmark ($k = 10$).

Let us discuss the effect of increasing k on `JOSIE-D`, as it also reads candidates, similar to `ProbeSet-D`. The running time of `JOSIE-D` increases with k, but at a much slower rate than `ProbeSet-D`. This is because unlike the latter, `JOSIE-D` does not read all the candidates it encounters – it always reads the next candidate with the lowest net cost (and likely carrying better pruning power, or benefit) evaluated by the cost model, and aggressively prunes out unqualified candidates using the position filter.

We compare the mean number of sets read by `ProbeSet-D` and `JOSIE-D` per query in the second row of Figure 3.6. It is evident that `JOSIE-D` drastically reduced the sets read by an order of magnitude (from well over 100 to around 30). This shows that the cost model is extremely effective in choosing the best next candidate to read, which causes much more aggressive pruning. We also compare the mean number of posting lists read in the third row of Figure 3.6, which shows that `JOSIE-D` reads more posting lists than `ProbeSet-D`. This is because the net cost of reading posting lists may be lower than reading the next best candidates, as discussed in Section 3.3.3.

What happened at $k = 20$ on the 1k benchmark? The estimation time becomes a major fraction of the total time when read time is small – this happens when the candidates are small in size, but large in total number as happens with small queries at large k. The estimation time is linear in the total number

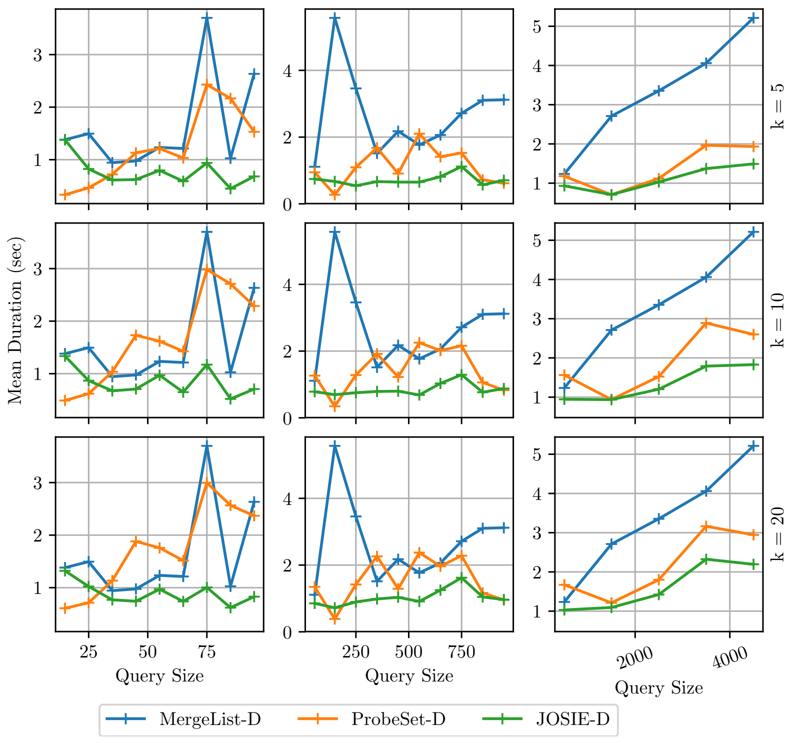


Figure 3.7: Mean query time on WebTable benchmark.

of candidates, as `JOSIE-D` must always go through all unread candidates. In this case, `MergeList-D` may benefit from the fact that it does not read any candidate, and the number of posting lists to read is also small for small queries.

We also compare the memory footprint of query processing for the three algorithms, in the fourth row of Figure 3.6. For the detail of memory footprint measurement, please refer to Section 3.5.8. Based on the results, `ProbeSet-D` uses the most memory, and `JOSIE-D` comes in second. This is due to the large sets in the Open Data benchmark, as the two algorithms both need to allocate buffers for reading candidate sets. The relatively short posting lists in the Open Data benchmark also lead to the low memory usage of `MergeList-D`.

3.5.6 Webtable Results

In comparison with Open Data, Web Tables has different characteristics – it has 219× more sets, but much smaller sets (average size 10 versus 1,540), and its tokens have higher frequencies overall (average 4 versus 23), leading to larger posting lists that are more expensive to read.

The effect of large posting lists directly impacts the performance of `MergeList-D`, which must read all distinct posting lists for a query set. As shown in Figure 3.7, it has the longest running time, and can be 3× slower than `ProbeSet-D` and `JOSIE-D` on $k = 5$. On the other hand, `ProbeSet-D` benefits from the small set sizes in Web Tables, as the time saved from reading fewer large posting lists is larger than the time to read tiny sets. As a consequence, for table repositories with similar distributions to the Web Table benchmark – high token frequencies and small set sizes – search algorithms making use of prefix filter (e.g., `ProbeSet-D` and `JOSIE-D`) would benefit the most from the distribution.

What about `JOSIE-D`? It still out-performs the other two algorithms, as shown in Figure 3.7, with

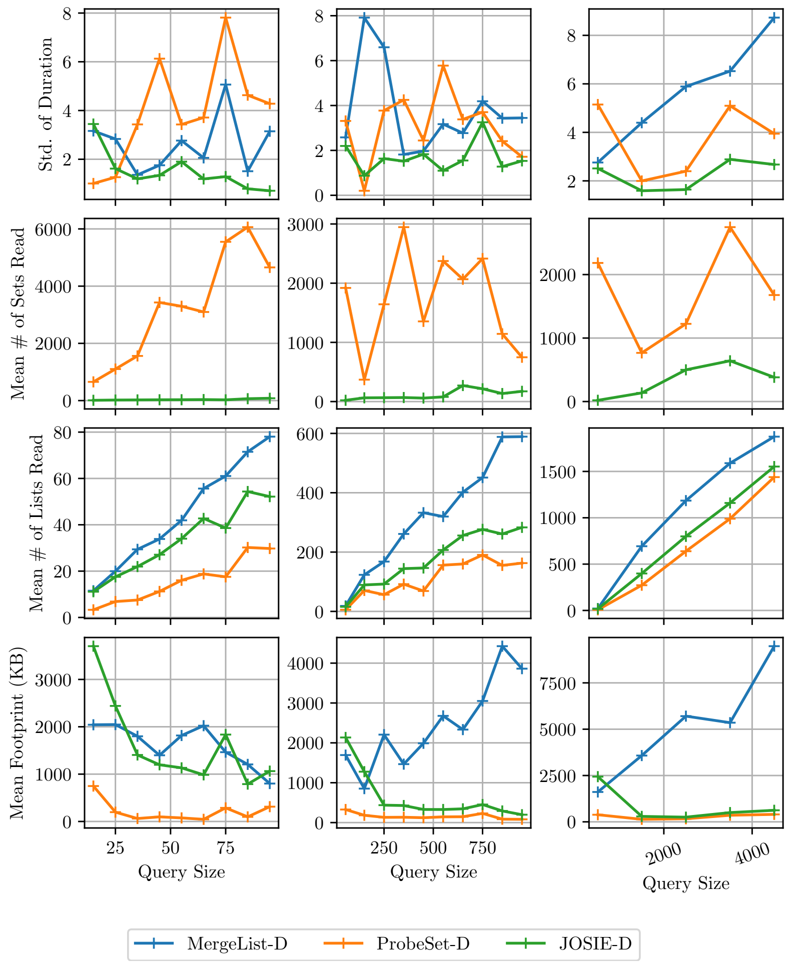


Figure 3.8: The standard deviation, number of sets and posting lists read, and query memory footprint on Web Table benchmark ($k = 10$).

lower variance as shown in the first row of Figure 3.8. The most interesting part is that it still mostly out-performs `ProbeSet-D` despite the high estimation time caused by having many candidates to look through. The second row of Figure 3.8 shows that the number of candidates in Web Tables is much higher than Open Data, approximately 15× on the 1k benchmark, for example. `JOSIE-D` still reads an order of magnitude fewer candidates than `ProbeSet-D`, so the estimation time is largely compensated for by the time saved from pruned candidates.

The fourth row of Figure 3.8 shows the memory footprint of the three algorithms on the Web Table benchmark. The result is very different from the Open Data benchmark: `MergeList-D` comes in first in memory usage, exceeding the other two by an order of magnitude for query sizes larger than 250. This is due to the longer posting lists in the Web Table benchmark, making it much more costly space-wise to read posting lists. The smaller set sizes reduce the memory usage of both `ProbeSet-D` and `JOSIE-D`, however, because the latter still reads more posting lists (δ more, as explained in Section 3.3.4), it tends to use more memory.

One interesting result is that `JOSIE-D` uses a lot more memory for smaller queries than larger ones. This is because the algorithm tends to read posting lists rather than sets for small queries, as shown in the second row of Figure 3.8, and posting lists require larger buffer than sets in this benchmark. `JOSIE-D`

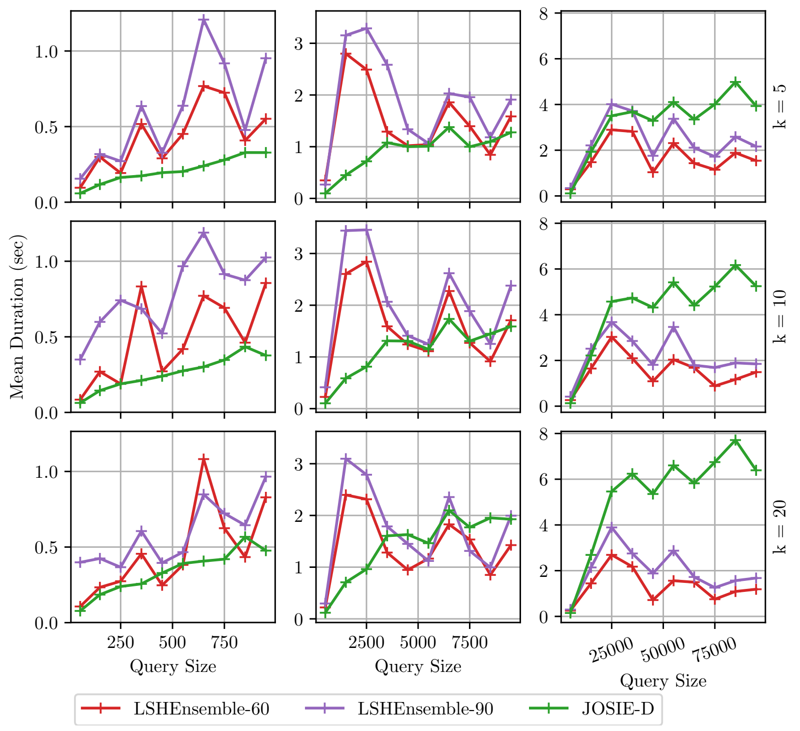


Figure 3.9: Mean query time of LSH Ensemble (32 partitions, modified for top-k) and JOSIE on Open Data benchmark.

also uses more memory than `MergeList-D` for those small queries, because unlike `MergeList-D`, it needs to read token positions and set sizes in addition to set IDs (see Section 3.5.8).

3.5.7 Comparison with LSH Ensemble

The most interesting comparison comes from `JOSIE-D` and LSH Ensemble algorithms (`LSHEnsemble-60` and `90`) using 32 partitions. Figure 3.9 shows the mean query durations. The LSH Ensemble query duration includes the time of retrieving candidates from the LSH index and the time to read and compute exact intersection sizes for candidates. The common key to the success of LSH algorithms is that the first stage of candidate retrieval is extremely fast and can be very selective, such that the total query duration can be lower than exact algorithms.

However, based on Figure 3.9, we observe that the performance of `JOSIE-D` can beat LSH Ensemble by 2× on the 1k benchmark for $k = 5$ to $k = 10$. This is surprising, because LSH Ensemble is an approximate algorithm (though we have modified it to achieve a minimum 60% recall for `LSHEnsemble-60` and 90% for `-90`) while `JOSIE-D` gives exact results. As query size increases, for example, on the 100k benchmark, LSH Ensemble begins to out-perform `JOSIE-D` by a large margin. This is also true with increasing k, as at $k = 20$ on the 100k benchmark, LSH Ensemble out-performs `JOSIE-D` by 3.5×.

The reason for `JOSIE-D` being faster at smaller k is actually related to observations made by Bayardo et al. who showed experimentally that prefix-filter-based exact techniques can be faster than LSH [5]. This is because the prefix filter can be much more selective than LSH index at high thresholds or small k, leading to very few posting list reads and smaller number of candidate reads.

Why is LSH Ensemble slower than `JOSIE-D` for small queries? Our investigation shows that this

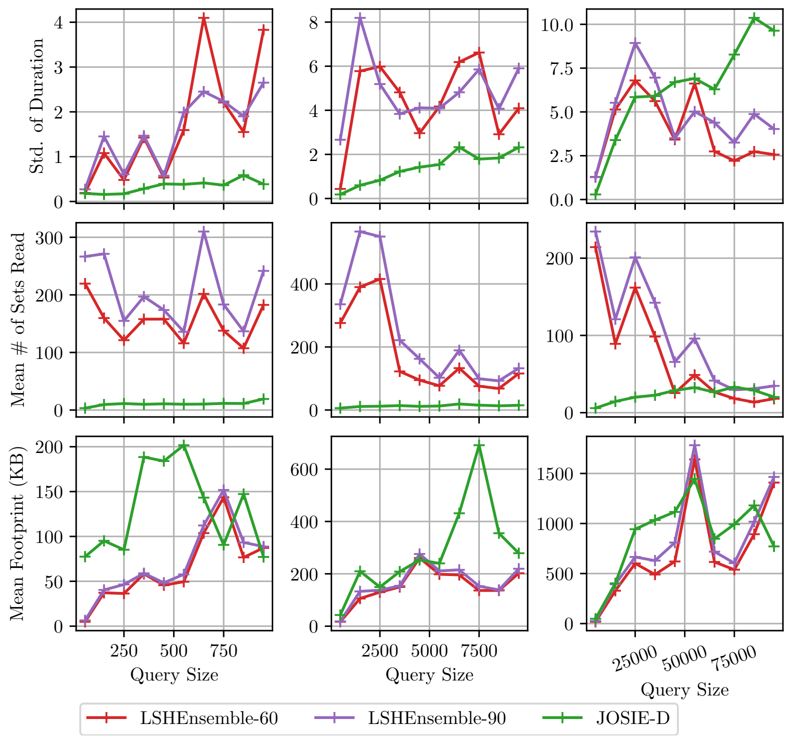


Figure 3.10: The standard deviation, number of sets read and query memory footprint on Open Data benchmark for $k = 10$.

is because the transformation from containment similarity (i.e., normalized intersection size $\frac{|Q \cap X|}{|Q|}$) to Jaccard similarity used by LSH Ensemble introduces additional false positive candidates [74], which causes wasted time spent in reading those sets. We found that this influx of false positive candidates is the most severe at small queries, due to the skewed set size distribution where there are many more small sets than large sets. A sudden drop in the number of candidates can be observed in the second row of Figure 3.10, the plot for the 10k benchmark. Unlike `JOSIE-D`, LSH Ensemble cannot use position filter to prune out false positive candidates, it must read all candidates it encounters. As shown in the 10k and 100k benchmarks of Figure 3.10, the number of candidates read by LSH Ensemble drops closer to that of `JOSIE-D`, while its running time becomes relatively faster than the latter. This is because LSH Ensemble does not need to read any posting lists, and the candidate retrieval time is purely in-memory and extremely fast, while `JOSIE-D` must issue read operations for posting lists to retrieve candidates.

The third row of Figure 3.10 compares the query memory footprint of LSH Ensemble and `JOSIE-D`. Because `JOSIE-D` needs to read posting lists in addition to sets, it is expected to use more memory than LSH Ensemble.

In conclusion, LSH Ensemble can be a better choice if the user wants to retrieve many results ($k \geq 20$) and the query size is large (more than 10k) and most importantly, if the user does not care very much about recall of the results. Otherwise, a cost-based exact algorithm such as `JOSIE-D` is the better choice.

3.5.8 Measuring Query Memory Footprint

Measuring memory footprint based on heap usage has to deal with noise such as overhead in programming language runtime, garbage collection, and various compiler optimizations. Thus, we only measure the amount of data structure allocated for the purpose of processing the query. In this section, we explain our approach to measure the query memory footprint of the algorithms in Section 3.5.4.

We use Equation 3.16 to calculate the memory footprint of `MergeList-D`. The first term is the size of a hash map for tracking the counts of times the candidates appear in posting lists. Its size is calculated using the number distinct sets encountered ($|W_{ML}|$), times the size of each set ID and count. The second term is the size of the buffer that is allocated to read posting lists. We use the longest posting list read to calculate the size. Since `MergeList-D` only needs the set IDs in each posting list, and does not need the token positions and set sizes, the buffer size is simply the total size of integer set IDs in the longest posting list.

$$|W_{ML}| \cdot 2 \cdot \textsc{SizeOf}(Int) + \max_{i \in [1,|Q|]} f_i \cdot \textsc{SizeOf}(Int) \tag{3.16}$$

We use Equation 3.17 to calculate the memory footprint of `ProbeSet-D`. The first term is the size of the buffer allocated for reading posting lists. Due to the use of prefix filter (subset of posting lists from 1 to p^*), the buffer size is the size of the longest posting list in the prefix. However, since `ProbeSet-D` needs the token positions and set sizes for position filter, in addition to set IDs, the size of each posting list entry is 3 integers. The second term is the size of the buffer allocated for reading sets. Similar to the other buffer, its size is calculated using the maximum size read. The third term is the size of the hash set allocated for tracking sets that have been pruned or read, so it is simply the size of the IDs of all the sets that appeared in the prefix posting lists. The set of all sets that appear in the prefix is W.

$$\begin{aligned}\max_{i \in [1,p^*]} 3f_i \cdot \textsc{SizeOf}(Int) + \max_{X \in W \setminus V} |X[j_{X,0}:]| \\ + |W| \cdot \textsc{SizeOf}(Int)\end{aligned} \tag{3.17}$$

Equation 3.18 is used to calculate the memory footprint for `JOSIE-D`, where δ is the extra posting lists read after the prefix filter, V^* is the set of pruned candidates using the position filter, and W_i is the set of candidates after reading posting list i. See Equation 3.14 and 3.7 for their usages in running time analysis. The first three terms have nearly indentical expression as those of `ProbeSet-D`, however the magnitudes can be different, as $p^* + \delta \geq p^*$, and $W \setminus V^* \subseteq W \setminus V$. The last term is the maximum size of the hash map allocated for unread candidates (W_i) after reading a batch of posting lists that end at i. As described in Section 3.3.2, in addition to the set ID, we keep track of 4 additional integers for each candidate. Also, since pruned candidates are removed from the hash map, we use the maximum count of candidate sets during query processing to calculate the allocated size, as the deleted slots can be reused.

$$\begin{aligned}\max_{i \in [1,p^*+\delta]} 3f_i \cdot \textsc{SizeOf}(Int) + \max_{X \in W \setminus V^*} |X[j_{X,0}:]| \\ + |W| \cdot \textsc{SizeOf}(Int) + \max_{i \in [1,p^*+\delta]} 5|W_i| \cdot \textsc{SizeOf}(Int)\end{aligned} \tag{3.18}$$

For the LSH Ensemble algorithms, `LSHEnsemble-60` and `LSHEnsemble-90`, we use Equation 3.19. The first term is the size of the buffer allocated for reading sets, which is the maximum size over all sets read. The second term is the size of the hash set allocated for tracking the candidates retrieved from the LSH indexes.

$$\max_{X \in W_{LSH}} |X| + |W_{LSH}| \cdot \textsc{SizeOf}(Int) \tag{3.19}$$

During the experiments, we record all the relevant variables needed to calculate the memory footprint. The experimental results on Open Data and Web Table benchmarks are shown in Figure 3.6, 3.8, and 3.10.

3.6 Related Work

The overlap set similarity search problem is a type of set similarity search where the similarity measure is set intersection size. A very similar problem is *set similarity join*, which is also know as all-pair set similarity search. Given a collection of sets, set similarity join finds all the set pairs whose similarity is less than a threshold. There are many works on in-memory set similarity join. Mann et al. [46] present an excellent experimental study. Specifically, Arasu et al. [1] designed a filtering condition with guaranteed false positive rate while not missing any result. Bayardo et al. [5] proposed to use prefix filter for set similarity join. Xiao et al. [70] extended the work by Bayardo et al. [5] with two additional filters: the position filter, which we have discussed extensively in Section 3.2.3, and the suffix filter. We do not use the suffix filter given the later Mann et al. study showed it was not very effective (something we confirmed in our initial development of JOSIE). Instead of using a fixed-length prefix as done by Bayardo et al., Wang et al. [65] designed an adaptive prefix filter framework. Wang et al. [67] further improved Xiao et al.'s work through sharing computation cost among candidate sets.

A lot of research on parallel set similarity join has been done [56, 64, 23, 57, 61, 49]. Fier et al. [28] present a recent experimental study of this work. Vernica et al. [64] designed a parallel algorithm based on the prefix filter. MassJoin [23] uses partition-based string similarity join [43]. ClusterJoin [57] partitions the sets and distributes the partitions to different nodes. The goal of these parallel approaches is to scale in the number of sets, not necessarily in the size of the sets. For example, in the comparative study of Fier et al., the maximum dictionary size is 8M.

In addition to threshold-based set similarity join algorithms, Xiao et al. [69] proposed an in-memory algorithm for top-k set similarity join with Jaccard/Cosine similarity measures. We have introduced the search version of this algorithm in Section 3.2.2. Deng et al. [22] and Wang et al. [66] studied the top-k string similarity search problem. Behm et al. [6] studied the string similarity search problem in external memory. Note that all these techniques are not designed for data lakes. They are typically evaluated on sets with small average size and small dictionaries.

There is also research on finding related (not necessarily joinable) tables in data lakes. Sarma et al. [58] use keyword search and similarity measures other than intersection size to find tables that are candidates for join and candidates for union from Web Tables [11]. The Mannheim Search Engine applies keyword search techniques to find joinable attributes by treating attribute values as keywords and ranking candidate attributes using a fuzzy Jaccard similarity [41]. This work is not intended to scale to large sets. Lehmberg et al. [39] found that stitching small web tables can help match them with knowledge bases. Nargesian et al. [51] proposed techniques to search for unionable tables from data lakes. Deng et al. [21] designed a set relatedness metric which can be used to find related tables.

Approximate set similarity search techniques have been used for finding related tables. Both Asymmetric Minwise Hashing [60] and LSH Ensemble [74] use MinHash-based indexing for set containment search. LSH Ensemble was shown to handle data sets with skewed cardinality distribution much better than Asymmetric Minwise Hashing, hence we have used LSH Ensemble in our experimental study in Section 3.5. Fernandez et al. [26] use MinHash LSH to find similar tables based on Jaccard similarity between columns. For the skewed distributions we consider, Jaccard is not an appropriate measure as we showed in Chapter 2.

3.7 Summary

We have presented `JOSIE`, a new exact top-k overlap set similarity search algorithm. Unlike previous approaches, `JOSIE` uses a cost model to adapt to the data distribution enabling significantly better performance over data lakes with large sets than the state-of-the-art approach `ProbeSet`. For queries with sizes up to 10K (a larger size than has been studied in the exact set similarity search literature), `JOSIE` even out-performs what is currently the best approximate approach, our own LSH Ensemble from Chapter 2. Our extensive experimental evaluation is the first to compare overlap set similarity search approaches over a repository with sets of average size over 1K and maximum size in the millions (two orders of magnitude larger than previous evaluations).

During the search, our approach estimates the likely intersection size between a candidate and the query. Going forward, we are considering how to improve the estimation of set intersection size in a way that takes into account the frequency of tokens. We believe this would lead to better estimation of the net cost and make our cost model more accurate.

Our work and our competitors for both exact and approximate search all use set intersection size to rank results. Of course, when searching relational tables, attributes may be bags. An open problem is to use bag semantics (including the multiplicities of tokens within a set) to rank results. This would allow a data scientist to be more informed about the relative usefulness of search results because it would reflect the actual size of the joined tables.

Chapter 4

Auto-Join: Smart Join with Transformations

In this chapter, we present Auto-Join, a powerful algorithm that automates joins with syntactic transformations. Tables in data lakes come from many different sources, which may have very different format conventions. The formatting difference makes it difficult to join tables from different sources. Auto-Join replaces the *ad hoc* manual script writing for making the format on the join columns consistent, with an automated process, making data scientists more efficient in using data lakes. This technique leverages substring indexes to efficiently identify joinable row pairs, from which it automatically learns *minimum-complexity* programs whose execution can lead to equi-joins. Auto-Join also scales to large tables while still maintaining interactive speed, using a sampling scheme that minimizes the number of rows sampled, which has a formal guarantee of success with high probability. This algorithm and the experimental evaluation presented in this chapter is published work [72].

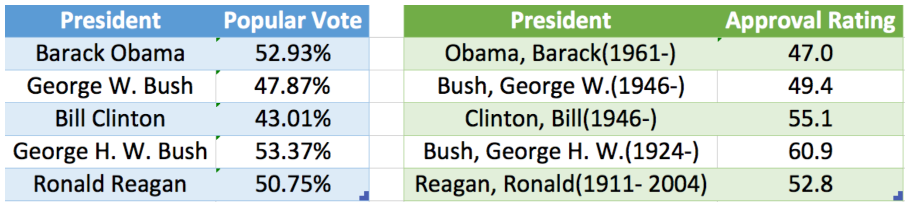
4.1 Equi-Join Is Not Enough

In Section 1.2, we have introduced the usefulness of the join operation for tables in data lakes, and established that join is essential to data analysis involving multiple tables. Many existing data analysis tools such as Power Query for Excel[1] and Informatica[2] support join operation.

Most existing commercial systems only support *equi-join* through exact equality comparisons. While equi-join works well in well-curated settings such as data warehousing, it may not be sufficient in less curated situations, such as in the case of data lakes. For example, analysts today increasingly need to perform *one-off*, *ad hoc* analyses by joining tables from different original sources, whose key columns are often formatted differently. Requiring support staff to perform extensive ETL (extract-transform-load) operations in such scenarios is often too slow and expensive. In fact, customer surveys from a Microsoft data preparation system suggest that automating join is a key feature requested by end-users. Thus, solving the auto-join problem would be an important step towards making data lakes easier to manage.

Figure 4.1 shows an example. An analyst has a table on the left in her spreadsheets about US presidents and popular votes they won in elections, and a table on the right, that has information

[1]`https://support.office.com/en-us/article/merge-queries-power-query-fd157620-5470-4c0f-b132-7ca2616d17f9`
[2]`https://www.informatica.com`

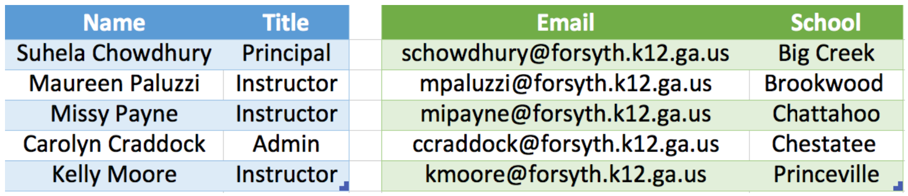


President	Popular Vote
Barack Obama	52.93%
George W. Bush	47.87%
Bill Clinton	43.01%
George H. W. Bush	53.37%
Ronald Reagan	50.75%

President	Approval Rating
Obama, Barack(1961-)	47.0
Bush, George W.(1946-)	49.4
Clinton, Bill(1946-)	55.1
Bush, George H. W.(1924-)	60.9
Reagan, Ronald(1911- 2004)	52.8

Figure 4.1: An example of joinable tables with different formats on the key columns. (left): US presidents and popular votes. (right): US presidents and job approval rating. The right table uses last-name, comma, first-name, with (year-of-birth and year-of-death).

about their job approval rating. Now she wants to join these two tables so that she can correlate them. However, the `name` columns of the two tables use different representations – the one on the left uses first-name followed by last-name, while the one on the right uses last-name, comma, first-name, with additional year-of-birth information in parenthesis. To date, the popular data analysis software such as Power Query for Excel[3] and Informatica[4] only support equi-join and would fail to work on these two tables. The analyst would have to either write her own transformation program so that the name representations become consistent for equi-join[5], or ask technology support staffs to perform such a transformation.



Name	Title
Suhela Chowdhury	Principal
Maureen Paluzzi	Instructor
Missy Payne	Instructor
Carolyn Craddock	Admin
Kelly Moore	Instructor

Email	School
schowdhury@forsyth.k12.ga.us	Big Creek
mpaluzzi@forsyth.k12.ga.us	Brookwood
mipayne@forsyth.k12.ga.us	Chattahoo
ccraddock@forsyth.k12.ga.us	Chestatee
kmoore@forsyth.k12.ga.us	Princeville

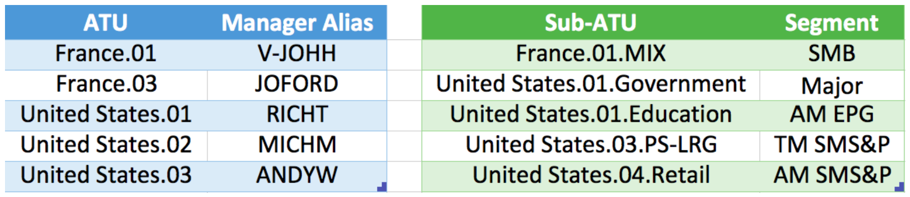
Figure 4.2: Another example of tables with different key column formats. (left): Name and job titles in school. (right): Email and school districts. `Email` can be generated from `name` in the left by concatenating first-initials, last-names, and `@forsynth.k12.ga.us`.

Figure 4.2 shows another case with two real tables collected from the web, where the table on the left uses teachers' full names, while the table on the right only has email addresses as the key column. Note that in this case because email aliases can be generated by taking first-initials and concatenating with last names, there exists a clear join relationship between the two tables. Equi-join, however, would again fail to work in this case.

Scenarios like these are increasingly common in ad-hoc data analysis, especially when analysts need to bring in data from different sources, such as tables extracted from public web pages.

It is worth noting that this join problem exists also in enterprise tables such as Excel files. Figure 4.3 shows a pair of real spreadsheet tables from a corpus of Excel files crawled in a large IT company. The `ATU` (area-team-unit) column on the left can be joined with `Sub-ATU` on the right by taking the first two components of `Sub-ATU`, and then equi-join with `ATU` in a hierarchical N:1 manner. Figure 4.4 shows another example from enterprise spreadsheets. The two tables cannot be equi-joined directly. However, if we concatenate `id` and `session name` in the left table with appropriate brackets, the two tables can then be equi-joined.

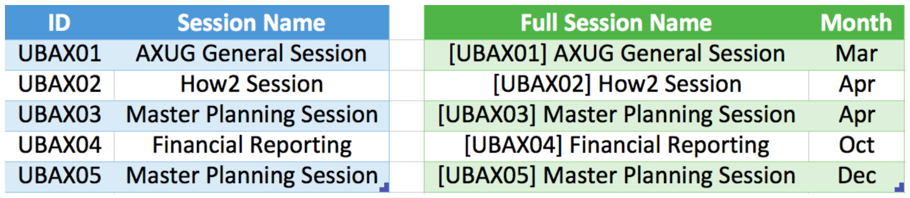
[3]`https://support.office.com/en-us/article/merge-queries-power-query-fd157620-5470-4c0f-b132-7ca2616d17f9`
[4]`https://www.informatica.com`
[5]Such behavior is observed in customer surveys and logs – for certain datasets users perform sequences of transformations in order to enable equi-join.



ATU	Manager Alias
France.01	V-JOHH
France.03	JOFORD
United States.01	RICHT
United States.02	MICHM
United States.03	ANDYW

Sub-ATU	Segment
France.01.MIX	SMB
United States.01.Government	Major
United States.01.Education	AM EPG
United States.03.PS-LRG	TM SMS&P
United States.04.Retail	AM SMS&P

Figure 4.3: An example of tables from enterprise spreadsheets. (left): `ATU` name (for area team unit). (right): `Sub-ATU` names organized under `ATU`.



ID	Session Name
UBAX01	AXUG General Session
UBAX02	How2 Session
UBAX03	Master Planning Session
UBAX04	Financial Reporting
UBAX05	Master Planning Session

Full Session Name	Month
[UBAX01] AXUG General Session	Mar
[UBAX02] How2 Session	Apr
[UBAX03] Master Planning Session	Apr
[UBAX04] Financial Reporting	Oct
[UBAX05] Master Planning Session	Dec

Figure 4.4: Another example of tables from enterprise spreadsheets. (left): `ID` and `session name` in separate fields. (right): Concatenated `full session name`.

Joining tables with different representations is a ubiquitous problem. Simple syntactic transformations (e.g., substring, split, concatenation) can often be applied to make equi-join possible. However, existing commercial systems can only perform equi-joins, and users often face two main challenges. (1) For big tables with a large number of rows and columns, manually identifying corresponding rows and columns from two tables that can join using transformations is non-trivial (i.e., manually searching sub-strings in large spreadsheets is slow and does not always produce hits). (2) End-users (e.g., Excel users) may not be able to program transformations to enable equi-join. In this chapter, our goal is to automate the discovery of syntactic transformations needed such that two tables with different representations can be joined with the click of a button. Note that because such transformations are driven by end-user tools, a significant technical challenge is to make such transformation-based join very efficient and at interactive speed.

4.2 Related Work

A straightforward approach is to ask users to provide transformation programs (either manually or with the help of example-driven tools like FlashFill [32] and Foofah [37]). These programs can then be used to produce derived columns, with which tables can be equi-joined. This is inconvenient for users – for large tables with many columns and millions of rows, the first task of identifying rows and columns from two tables with joinable values alone is non-trivial, as users often have to pick random sub-strings from one table to search for hits in the other, which is slow and often fails to produce matches. The need to find the "best" transformation in the context of joins further complicates the problem. We aim to automate this process so that users can join two *unordered sets* of rows with the click of a button – in comparison, we no longer require users to manually specify *matching pairs* of input/output examples as in FlashFill-like systems [32].

Since rows that join under syntactic transformations typically have substantial substring overlap, an alternative approach is to use fuzzy join [13]. The challenge is that fuzzy join has a wide range of parameters (tokenization, distance-function, thresholds, etc.) that need to be configured appropriately to work well. For example, fuzzy join can be configured to tokenize differently (tokenize by words, or by 2-gram, 3-gram, etc.), use different distance functions (Jaccard, Cosine, etc.), and different distance thresholds. The ideal configuration can vary significantly from case to case, and is difficult for users to determine. For instance, for Figure 4.1, one should tokenize by words, but that tokenization will fail completely for Figure 4.2, which requires q-gram tokenization.

Furthermore, even when fuzzy join is configured perfectly, it may still produce incorrect results due to its fuzzy and imprecise nature. For example, in Figure 4.1, if we tokenize by words, and want to join `Ronald Reagan` and `Reagan, Ronald(1911-2004)`, then the threshold for Jaccard distance should be at least 0.66 (this pair has a distance of $1.0 - \frac{1}{3} = 0.66$). However, using a threshold of 0.66 will also join `George W. Bush` with `Bush, George H. W.(1924-)` (where the distance is $1.0 - \frac{3}{5} = 0.4$), which is incorrect. [6] The root cause here is that fuzzy join uses an imprecise representation and a simple threshold-based decision-boundary that is in practice, not always correct. On the other hand, there are many cases where regularity of structures in data values exists (e.g. Figure 4.1-4.4), and for those cases using consistent transformations for equi-join complements fuzzy-join by overcoming its shortcomings mentioned above.

Warren and Tompa [68] proposed a technique to translate schemas between database tables, which is applicable to joins and is the only published technique that we are aware of that can produce transformation-based joins given two tables. However, the types of transformations they considered are rather limited (e.g., no string-split and component based indexing), and as a result their approach is not expressive enough to handle many real join tasks we encountered. As we will show in our experiments, their approach can handle less than 30% of the join cases we collected.

4.3 Problem Overview

Our objective is to automate transformation-joins by generating the transformations that are needed for equi-joins. Specifically, we want to transform columns of one table via a sequence of string-based syntactic operations, such that the derived columns can be equi-joined with another table. Example 4.3.1 gives such an example.

Example 4.3.1. *In Figure 4.1, there exists a transformation whose application on the right table can lead to equi-joining with the left table. We illustrate the sequence of operations in the transformation using the first row* $\{$[*Obama, Barack(1961-)*], [*47.0*]$\}$ *as an example.*

1. *Input row X with two elements:*
   $\{$[*Obama, Barack(1961-)*], [*47.0*]$\}$
2. *Take the first item $X[0]$,* Split *by "(", produce Y:*
   $\{$[*Obama, Barack*], [*1961-)*]$\}$
3. *Take the first item $Y[0]$,* Split *by ",", produce Z:*
   $\{$[*Obama*], [*Barack*]$\}$
4. *Takes* Substring *[1:] from $Z[1]$, produce T:*
   [*Barack*]
5. Concat *T, a constant string " " and $Z[0]$, produce*
   [*Barack Obama*]

[6]Notice that unlike in previous chapters, the literature in this area uses distance rather than similarity. Hence, we will require the distance to be less than a threshold rather than the similarity being over a threshold as before.

Table 4.1: Notations for analyzing q-gram matches.

Symbol	Description
T_s, T_t	T_s is the source table, T_t is the target table.
R_s, R_t	A row in T_s and T_t, respectively
C_s, C_t	A column in T_s and T_t, respectively
$\mathcal{Q}_q(v)$	The q-grams of a string value v
$\mathcal{Q}_q(C_s)$	The multi-set of all q-grams in C_s
$\mathcal{Q}_q(C_t)$	The multi-set of all q-grams in C_t

This derived value can then be equi-joined with the first row in the left table in Figure 4.1. It can be verified that the same transformation can also be applied on other rows in the right table to equi-join with the `President` *column of the left table.*

As discussed earlier, identifying joinable rows from large tables alone is non-trivial, not to mention programming transformations. We would like to automate this problem, which we call the *transformation join problem.*

Definition 4.3.1. *Transformation Join Problem: Given two tables T_s, T_t, and a predefined set of operators Ω, find a transformation $P = o_1 \cdot o_2 \cdot o_3 \cdot \ldots o_n$, using operators $o_i \in \Omega$, such that $P(T_s)$ can equi-join with key columns of T_t.*

Here each transformation P is composed of a sequence of operators in Ω, where the output of one operator is the input of the next. For the purpose of transformation-join, we have identified a small set of operators that are sufficient for almost all join scenarios we encountered.

$$\Omega = \{\textsc{Split},\ \textsc{Concat},\ \textsc{Substring},\ \textsc{Constant},\ \textsc{SelectK}\} \tag{4.1}$$

This set of operators Ω can be expanded to handle additional requirements as needed.

In this definition, because we require $P(T_s)$ to equi-join with key columns of T_t, the types of join we consider are implicitly constrained to be 1:1 join (key:key) and N:1 join (foreign-key:key). This constraint is important because it ensures that the joins we automatically generate are likely to be useful; relaxing this constraint often leads to N:M joins that are false positives (e.g., join by taking three random characters from two columns of values).

Also observe that we apply transformations on one table T_s in order to equi-join with another table T_t. We refer to the table T_s as the *source table*, and T_t as the *target table*, respectively.

Because in our problem, we are only given two tables with no prior knowledge of which table is the source and which is the target, we try to generate transformations in both directions. In Example 4.3.1 for instance, the right table is used as the source table. Changing direction in this case does not generate a transformation-join because year-of-birth is not present in the left table. Advanced handling of composite columns will be discussed in Section 4.4.3.

Solution Overview Our system has three main steps.

Step 1: Find Joinable Row Pairs. In our problem, we only take two tables as input, without knowing which row from T_s should join with which row from T_t. Generating transformations without such knowledge would be exceedingly slow, due to the quadratic combinations of rows in two tables that can potentially join.

So in the first stage, we attempt to "guess" the pairs of rows from the two tables that can potentially join. We leverage the observation that unique q-grams are indicative of possible join relationships (e.g. `Obama`), and develop an efficient search algorithm for joinable row pairs.

Step 2: Learn Transformation. Once we obtain enough row pairs that can potentially join, we learn a transformation that uses rows from T_s as input and generates output that can equi-join with key columns of T_t. In Example 4.3.1 for instance, the desired transformation uses $\{$[`Obama, Barack (1961-)`], [`47.0`] $\}$ as input, and produces [`Barack Obama`] as output to join with the key column of the left table. Since there likely exists many possible transformations for one particular input/output row pair, we use multiple examples to reduce the space of feasible transformations, and then pick the one with minimum complexity that likely best generalizes the observed examples. This learning process is repeated many times using different combinations of row pairs, and the transformation that joins the largest fraction of rows in T_t is produced as the result.

Step 3: Constrained Fuzzy Join. In certain cases such as tables on the web, the input tables may have inconsistent value representations or dirty values. For example, in Figure 4.2, the second row of the right table uses `mipayne@forsyth.k12.ga.us`, instead of first-initial concatenated by last-name like other rows (which would produce `mpayne@forsyth.k12.ga.us`). Thus, transformations may miss this joinable row pair. As an additional step to improve recall, we develop a mechanism that automatically finds a fuzzy-join with optimized configuration to maximize additional rows to join, without breaking the join cardinality constraints (i.e., 1:1 or N:1). This improves the coverage of joins on dirty tables, and is of independent interest for the important problem of automatically optimizing fuzzy join.

4.4 Auto-Join by Transformations

In this section, we discuss the first two steps of Auto-Join: (1) finding joinable row pairs, and (2) learning transformations that generalize the examples observed in these row pairs.

4.4.1 Finding Joinable Row Pairs

Let P be the desired transformation, such that $P(T_s)$ can equi-join with T_t as in Definition 4.3.1. Pairs of rows that join under this transformation P is termed as *joinable row pair*. For instance, the first row from the left and right table in Figure 4.1 is a joinable row pair.

Since users do not provide joinable row pair as input to our system (which is cumbersome to provide especially in large tables), in this section we explain our approach for "guessing" joinable row pairs candidates from the two table through unique q-grams matches. Note that finding such row pairs is important, because trying the quadratic number of row combinations exhaustively is too computationally expensive to be interactive.

We leverage the observation that the set of operations Ω considered for transformation-join (Equation 4.1) tend to preserve local *q-gram*. A q-gram [10] of a string v is a substring of v with q consecutive characters. A complete q-gram tokenization of v, denoted as $\mathcal{Q}_q(v)$, is the set of all possible q-grams of

v. For example:

$$\mathcal{Q}_5(\texttt{Database}) = \{\texttt{Datab}, \texttt{ataba}, \texttt{tabas}, \texttt{abase}\}$$

q-grams have been widely used for string similarity and language modeling, among other applications.

Operations required for transformation-joins in Ω all tend to preserve sequences of local q-grams, which is the key property we exploit to find joinable row pairs.

1-to-1 q-Gram Match

Intuitively, if we can find a *unique* q-gram that only occurs once in T_s and T_t, then this pair of rows is very likely to be a joinable row pair (e.g., q-gram `Barack` in Figure 4.1). We start by discussing such 1-to-1 matches, and show why they are likely joinable row pairs using a probabilistic argument.

It is known that q-grams in texts generally follows power-law model [7, 29].

$$p_q(k) = \frac{\frac{1}{k^{s_q}}}{\sum_{z=1}^{N} \frac{1}{z^{s_q}}} \tag{4.2}$$

Here k is the rank of a q-gram by frequency, N is the total number of q-grams, and s_q is a constant for a given q.

We have conducted experiments that count q-gram occurrences from over 100M Web tables extracted from a 2018 snapshot of the documents indexed by the Bing search engine (called Bing Web Tables to distinguish them from the WDC corpus used in previous chapters). We did the same for all tables in Wiki tables. We test whether q-grams and natural words in the large table corpus also follow power laws by plotting a rank vs. frequency graph as shown in Figure 4.5. Note that both axes of the plots are in log scale. The close-to-linear relationships observed on these plots (especially for words and q-grams with larger q) suggest that this is indeed the case.

Given a pair of tables whose q-grams are randomly drawn from such a power-law distribution, we can show that it is extremely unlikely that a q-gram appears exactly once in both tables *by chance* (for reasonably large tables, e.g., $N > 100$).

Proposition 4.4.1. *Given two columns C_s and C_t from tables T_s and T_t respectively, each with N q-grams from an alphabet of size $|\Sigma|$ that follow the power-law distribution above. The probability that a q-gram appears exactly once in both C_s and C_t by chance is bounded from above by*

$$\sum_{k=1}^{|\Sigma|^q} \left((1 - p_q(k))^{N-1} \cdot p_q(k)\right)^2 \tag{4.3}$$

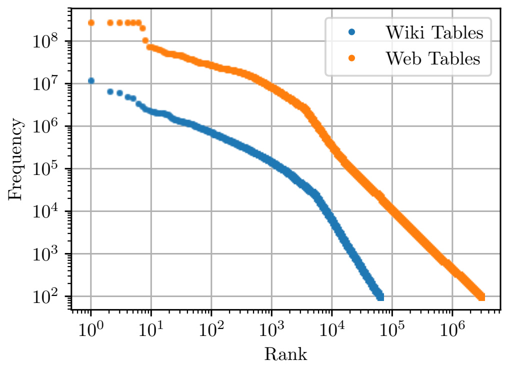
Proof. Given Equation 4.2, the probability that a q-gram with rank k appears exactly once in a random collection of N q-grams follows a geometric distribution:

$$(1 - p_q(k))^{N-1} \cdot p_q(k)$$

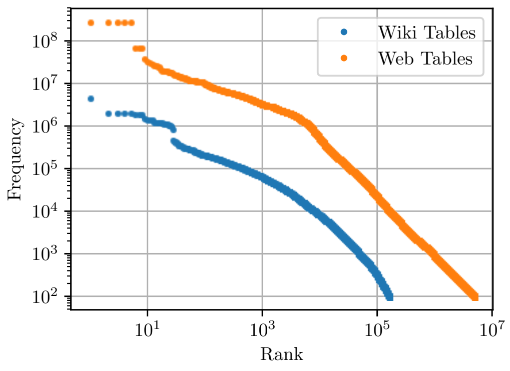
The probability of this q-gram appears exactly once each in two independent collections with N q-grams is then:

$$\left((1 - p_q(k))^{N-1} \cdot p_q(k)\right)^2$$

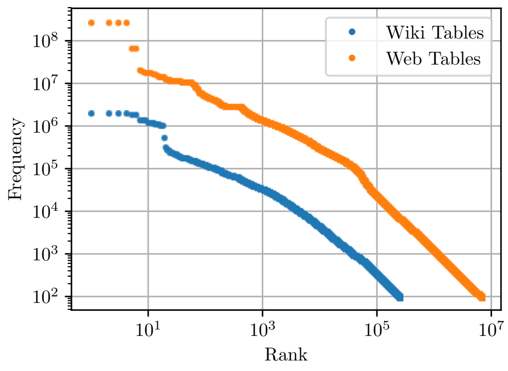
Lastly, the probability that at least one q-gram appears exactly once each in two collections can be



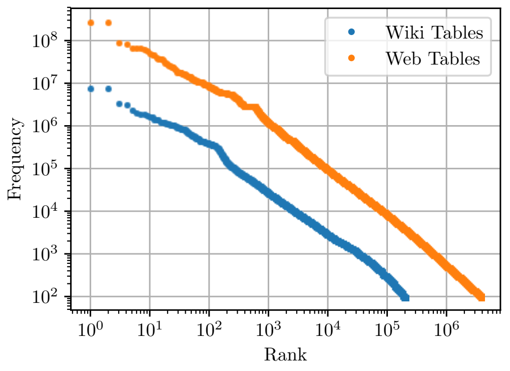
(a) Tokenization by 3-grams



(b) Tokenization by 4-grams



(c) Tokenization by 5-grams



(d) Tokenization by words

Figure 4.5: Frequency vs. rank in log-log plots. Close-to-linear relationships in the plots show that q-grams in Bing Web tables and Wikipedia tables also follow power laws, similar to what people have observed in general natural language text [7, 29].

computed using a summation over all possible $|\Sigma|^q$ q-grams:

$$\sum_{k=1}^{|\Sigma|^q} \left((1 - p_q(k))^{N-1} \cdot p_q(k) \right)^2$$

$\square$

For $q = 6$, $N = 100$, $|\Sigma| = 52$, and using the s_q defined in [7], the probability of any 6-gram appearing exactly once by chance on both columns is very small (< 0.00017). This quantity will in fact grow exponentially small for larger N (typical tables have at least thousands q-grams).

Given this result, we can conclude that if we do encounter unique 1-to-1 q-gram matches from two tables, they are unlikely coincidence but the result of certain relationships.

Let $\mathcal{Q}_q(C)$ be the multi-set of all the q-grams of distinct values [7] in column C; and let $F_q(g, C)$ be the number of occurrences of a q-gram $g \in \mathcal{Q}_q(C)$. Let v_s and v_t be the cell value at row R_s column C_s in T_s and row R_t column C_t in T_t, respectively. We define *1-to-1 q-gram matches* as follows.

Definition 4.4.1. *Let g be a q-gram with $g \in \mathcal{Q}_q(v_s)$ and $g \in \mathcal{Q}_q(v_t)$. If $F_q(g, C_s) = 1$ and $F_q(g, C_t) = 1$,*

[7]We remove possible duplicates in T_s columns since they are potentially foreign keys.

then g is a 1-to-1 q-gram match between row pair R_s and R_t with respect to the pair of column C_s and C_t.

As we have discussed, matches that are 1-to-1 q-gram are likely joinable row pairs.

Example 4.4.1. *Given two tables in Figure 4.1, the 6-gram `Barack` appears only once in both tables, and the corresponding rows in these two tables are indeed a joinable row pair. The same is true for q-grams like `chowdury` in Figure 4.2, `France.01` in Figure 4.3 and `UBAX01` in Figure 4.4, etc.*

As the reader will see in the experimental results (Section 4.7.3), using 1-to-1 q-gram matches as joining row pairs leads to a precision of 95.6% in a real-world benchmark.

General n-to-m q-Gram Match

Q-gram matches that are 1-to-1 are desirable special cases. In general we have *n-to-m q-gram matches.*

Definition 4.4.2. *Let g be a q-gram with $F(g, C_s) = n \geq 1$ and $F(g, C_t) = m \geq 1$, then g is a n-to-m q-gram match for corresponding rows with respect to the pair of column C_s and C_t.*

Compared to 1-to-1 q-gram matches that almost always identify a joinable row pair, the probability that a row pair identified by n-to-m matches is truly joinable is roughly $\frac{1}{nm}$. We use $\frac{1}{nm}$ to quantify the "goodness" of matches.

Note that the ideal q to identify n-to-m matches with small n and m can vary significantly in different cases.

Example 4.4.2. *For the tables in Figure 4.1, if we use 6-grams for value `Barack Obama`, we get an ideal 1-to-1 match of `Barack` between the first rows of these two tables. However, if we also use 6-gram for the second row `George W. Bush`, then the best we could generate is a 2-to-2 match using the 6-gram `George`, between the second and fourth rows of these two tables, respectively.*

For `George W. Bush`, the ideal q should be 9, since the 9-gram `George W.` could produce a 1-to-1 match. However, if we use 9-grams for the first row `Barack Obama`, we would fail to generate any q-gram match.

The ideal q is not known *a priori* and needs to be searched.

Efficient Search of q-Gram Matches

A simple algorithm for finding ideal q-gram matches (with small n and m) would conceptually operate as follows: (1) for every cell from one table, (2) for all possible settings of q, (3) for each q-gram in the resulting tokenization, (4) iterate through all values in the this table to find the number of q-gram matches, denoted as n; (5) iterate through all values in the other table to find the number of q-gram matches, denoted as m. The resulting match can be declared as an n-to-m match. This is clearly inefficient and would fail to make the desired join interactive. In this section we describe efficient techniques we use to search for unique q-gram matches.

First, we build a *suffix array index* [45] for every column in the source table and each column of the target table, so that instead of using step (4) and (5) above, we can search with logarithmic complexity. A suffix array index is built by creating a sorted array of the suffixes of all values in a column. Given a query q-gram, matches can be found by using binary search over the sorted array and looking for prefixes of the suffixes that match the query exactly. The complexity of probing a q-gram in the index is

$O(\log S)$, where S is the number of unique suffixes. We probe each q-gram once in both tables, to find the number of matches n and m.

Using suffix array significantly improves search efficiency for a given q-gram. However, for a cell value v, we still need to test all possible q-grams. To efficiently find the best q-gram (with the highest $\frac{1}{nm}$ score), we express the optimal q-gram g^* as the best prefix of all possible suffixes of v.

$$g^* = \operatorname*{arg\,max}_{\forall g \in \textsc{Prefixes}(u), u \in \textsc{Suffixes}(v)} \frac{1}{nm} \tag{4.4}$$

Where $n = F(g, C_s) > 0$ and $m = F(g, C_t) > 0$ are the number of matches in column C_s and C_t, respectively. We leverage a monotonicity property described below.

Proposition 4.4.2. *Let g_u^q be a prefix of a suffix u with length q. As the length increases by 1 and g_u^q extends at the end, the $\frac{1}{nm}$ score of the longer prefix g_u^{q+1} is monotonically non-increasing, or $F(g_u^{q+1}, C_s) \leq F(g_u^q, C_s)$ and $F(g_u^{q+1}, C_t) \leq F(g_u^q, C_t)$.*

Proof. Let a $g_u^q = x_1 x_2 \dots x_q$ be a prefix with length q, and $S = s_1 s_2 \dots s_n$ be a string of length n, where $x_1, x_2, \dots$ and $s_1, s_2, \dots$ are characters. Let $S[i:j]$ be a slice of S from s_i to s_j, where $1 \leq i < j \leq n$. If $S[i:j] = g_u^q$, then $S[i:j-1] = g_u^q[1:q-1]$. Thus, if g_u^q has one match with S, then $g_u^q[1:q-1]$ must have at least one match with S. Reversely, if $S[i:j-1] = g_u^q[1:q-1]$ but $s_j \neq a_q$, then $S[i:j] \neq g_u^q$. Thus, if $g_u^q[1:q-1]$ has a match with S, then g_u^q may not have a match with S. Therefore, the number of matches for a prefix g_u^q with a string S monotonically decreases as q increases. □

Given Proposition 4.4.2, for every suffix u we can find $g_u^{q^*}$ by looking for the longest prefix with matches in C_t using binary search. The global optimal g^* can be found by taking the $g_u^{q^*}$ with the highest score for all u. In practice, we found that when $q < 3$, the number of q-gram matches can be very large, severely impacting performance. Thus, we force q and the minimum length of suffixes to be at least 3.

Example 4.4.3. *In Figure 4.1, for the value `George W. Bush`, we iterate through all its suffixes (e.g., "`George W. Bush`", "`eorge W. Bush`", etc.). For each suffix, we test their prefixes using binary search to find the one with the best score (the longest prefix with match), from which we select the best prefix. In this case the prefix "`George W.`" for the first suffix is the best g^*.*

With this 1-to-1 match, we can determine that the first rows from the left/right tables in Figure 4.1 are joinable row pairs. Similarly the second rows from the two tables are also joinable row pairs, etc.

Because of the use of suffix array indexes and binary search, our overall search complexity is $O(|v| \log |v| \log S)$, which is orders of magnitude more efficient than the simple method discussed at the beginning of this section.

Putting it together: Find Joinable Rows

Algorithm 3 shows detailed pseudo code for finding joinable row pairs. $\textsc{KeyColumns}(T)$ returns all the single columns that is part of a key column in the table T. $\textsc{Suffixes}(v)$ returns all suffixes of a value v. $\textsc{QueryIndex}(C, g)$ uses a suffix array index built for the column C, and returns a list of rows containing g. The suffix array index can cache query results to efficiently serve queries that have been seen queried. MAXBYSCORE returns the optimal row pairs with the highest score. Note that more than one row pairs may have the same highest score.

In Section 4.4.1, we state that the optimal prefix $g_u^{q^*}$ for a suffix u is the one that has the highest $\frac{1}{nm}$ score. Here we present our algorithm for finding $g_u^{q^*}$. First, due to the monotonicity property given by Proposition 4.4.2, the value q^* that leads to the highest score should results in the smallest possible non-zero number of matches in C_t. That is, let r_t be the matching rows in C_t returned by QUERYINDEX, calling QUERYINDEX$(C_t, g_u^{q^*})$ results in $|r_t| \geq 1$, while calling QUERYINDEX$(C_t, g_u^{q^*+1})$ results in $|r_t| = 0$. The source column C_s is not needed in finding $g_u^{q^*}$. This is because (1) there is always a match in the source column (i.e., the current row itself) so $n = |r_s| > 0$ always, and (2) when $m = |r_t| = 0$, the score $\frac{1}{nm}$ becomes undefined and thus the corresponding q is infeasible. Therefore, q^* is only obtained at the conditions mentioned above, and is not dependent on C_s. BINARYSEARCHQ(u, C) performs the binary search of q^* that finds $g_u^{q^*}$ in C.

In practice, we found that when $q < 3$, the number of q-gram matches can be very large, severely impacting performance. Thus, we force q and the minimum length of suffixes to be at least 3.

It is worth noting that we group row pairs by the column pairs from which the matches are found. This is because we want to produce consistent transformations on certain columns C_s in T_s, so that the results can equi-join with columns C_t in T_t. As such, matches found in different column pairs are good indications that they belong to different transformation-join relationships, as illustrated by the following example.

Example 4.4.4. *In Figure 4.2, in addition to q-gram matches between the columns `Name` and `Email`, there is a q-gram match for `Princ`, which matches `Principal` in the first row in the `Title` column from the left table, and `Princeville` in the last row in the `School` column from the right table. However, matching row pairs produced between `Title` and `School` can be used to produce a transformation-join relationship between these columns (if one exists), which should be treated separately from one produced using matches between the `Name` and `Email` columns.*

Algorithm 3 Complete pseudo code for joinable row pair

```
function FindJoinableRowPairs($T_s$, $T_t$)
    $M \leftarrow \{\}$                                        ▷ $q$-gram matches
    for all $C_s \in T_s$ do
        for all $C_t \in$ KeyColumns($T_t$) do
            for all $v \in C_s$ do
                $m \leftarrow \{\}$
                for all $u \in$ Suffixes($v$) do
                    $q^* \leftarrow$ BinarySearchQ($u, C_t$)
                    if $q^* < 3$ or $q^* >$ Length($u$) then
                        continue
                    end if
                    $g_u^{q^*} \leftarrow u[1 : q^*]$
                    $r_s \leftarrow$ QueryIndex($C_s, g_u^{q^*}$)
                    $r_t \leftarrow$ QueryIndex($C_t, g_u^{q^*}$)
                    for all $(R_s, R_t) \in$ Pairs($r_s, r_t$) do
                        $m \leftarrow \cup\{(g_u^{q^*}, R_s, R_t, \frac{1}{|r_s||r_t|})\}$
                    end for
                end for
                $(g^*, R_s^*, R_t^*, score) \cdots \leftarrow$ MaxByScore($m$)
                $M \leftarrow \cup\{(g^*, (R_s^*, R_t^*), (C_s, C_t), score) \ldots\}$
            end for
        end for
    end for
    return SortByScore($M$), GroupBy($(C_s, C_t)$)
end function
function BinarySearchQ($u, C_t$)
    $a \leftarrow 3$, $b \leftarrow$ Length($u$) $+ 1$
    while $a < b$ do
        $h \leftarrow a + (b - a)/2$
        $r_t \leftarrow$ QueryIndex($C_t, u[1 : h]$)
        if $|r_t| > 0$ then
            $a \leftarrow h + 1$
        else
            $b \leftarrow h$
        end if
    end while
    return $a - 1$                              ▷ $a$ is the smallest $q$ for $|r_t| = 0$
end function
```

4.4.2 Transformation Learning

Given joinable row pairs $\{(R_s, R_t)\}$ produced for some column pair C_s and C_t from the previous step, we will now generate transformation programs using these pairs as examples. Specifically, we can view row R_s from T_s as input to a transformation program, and R_t projected on some key columns K of T_t as the desired output. If a transformation can take R_s as input and produce key columns K of R_t, equi-join becomes possible.

Physical Operators

Recall that we generate transformations using the following set of physical operators,

$$\Omega = \{\textsc{Split}, \textsc{SelectK}, \textsc{Concat}, \textsc{Substring}, \textsc{Constant}\}$$

The detailed interface of each operator is as follows.

- `string[] Split(string v, string sep)`
- `string SelectK(string[] array, int k)`
- `string Concat(string u, string v)`
- `string Constant(string v)`
- `string Substring(string v, int start, int length, Casing c)`

Each operator is quite self-explanatory. Split splits an input string using separator; SelectK selects the k-th element from an array; Concat performs concatenation; Constant produces a constant string; and finally Substring returns a substring from a starting index position (counting forward or backward) for a fixed length, with appropriate casing (lower case, upper case, title case, etc.).

In designing the operator space for Auto-Join, we referred to the string transformation primitives defined in the spec of the `String` class of C# and Java. The physical operators we use are a core subset of the built-in `String` functions of these languages.

Disambiguate Transformations by Examples

While in Example 4.3.1 we illustrate transformations using one joinable row pair for simplicity, in practice with only one row pair there often exists multiple plausible transformations.

Example 4.4.5. *In Example 4.3.1 the input row X has two elements $\{[$Obama, Barack(1961-)$]$, $[$47.0$]\}$, and the target output is $[$Barack Obama$]$. In addition to the transformation shown in that example, an alternative transformations that can also produce this output is:*

1. *Take the first item $X[0]$, Substr[8:6], produce $[$Barack$]$*
2. *Concat with constant string " ", produce $[$Barack $]$*
3. *Concat with $X[0]$, Substr[0:5], to produce the target output $[$Barack Obama$]$*

There exists many candidate transformations given only one input/output example pair. However, most of transformations would fail to generalize to other example pairs. The observation here is that if we use multiple joinable row pairs as input/output examples, the space of possible transformations are significantly constrained, such that the incorrect transformations will be pruned out. For example, if we just add the second rows from Figure 4.1 as an example pair, with $\{[$`Bush, George W.(1946-)`$]$, $[$`49.4`$]\}$

as the input and $\big[$`George W. Bush`$\big]$ as the output, then the transformation discussed in Example 4.4.5 would no longer be valid, as it would produce $\big[$`eorge Bush,`$\big]$, which cannot be joined with the keys in the other row.

The pruning power grows exponentially with the number of examples (details of this analysis can be found in Section 5.4 of the Appendix). In practice we just need a few examples (3 or 4) to constrain the space of candidate programs enough and generate the desired transformations. Note that we choose to use a small number of examples, because if we use hundreds of row pairs as examples, if even one row pair is a false-positive and not actually joinable, no consistent transformation can be produced and the the learning step would likely fail.

Learning via Logical Operators

The learning problem now is to find consistent transformations for a small set of input/output example row pairs. While the execution of transformations can be decomposed into simple physical operators defined in Ω, these are too fine-grained and do not directly correspond to our logical view of the transformation steps that humans would take. For instance, in Example 4.1 when we use $\big\{\big[$`Obama, Barack(1961-)`$\big]$, $\big[$`47.0`$\big]\big\}$ as input to produce $\big[$`Barack Obama`$\big]$ as output, humans would naturally view the required transformation as having three distinct logical steps – extract the component `Barack`, produce a space " ", extract the component `Obama`. Note that these logical operations correspond to a higher-level view that can always translate into a combination of simple physical operators – extracting the first of component implemented as Split by "(" followed by Split by ",", and finally a Substring.

For the technical reason of learning programs from examples, by mimicking how humans rationalize transformations, we introduce a set of higher-level *logical operators* Θ, each of which can be written as a sequence of physical operators. Θ={Constant, Substr, SplitSubstr, SplitSplitSubstr}

Unlike physical operators, each logical operator always returns a string. Each logical operator can be viewed as a "step" that contributes "progress" (partial output) to final results. It is important that logical operators all return strings, so that during automatic program generation, at each step we can decide which logical operator is more promising based on "progress". In comparison, physical operators like SelectK often need to be used in conjunction with other operators like SubStr before producing partial output, thus not directly amenable to automatic program generation.

Here we give a specification of logical operators using physical operators.

`string SplitSubstr(string[] array, int k, string sep, int m, int start, int length, Casing c) := Substring(SelectK(Split(SelectK(array, k), sep), m), start, length, c)`

`string SplitSplitSubstr(string[] array, int k`$_1$`, string sep`$_1$`, int k`$_2$`, string sep`$_2$`, int m, int start, int length, Casing c) := Substring(SelectK(Split(SelectK(Split( SelectK(array, k`$_1$`), sep`$_1$`), k`$_2$`), sep`$_2$`), m), start, length, c)`

`string Constant(string input) := input`

`string Substring(string[] array, int m, int start, int length, Casing c) := Substring(array, m, start, length, c)`

Note that Concat is a physical operator but not defined as a logical operator above. When programs are generated by the learning procedure, Concat is used implicitly to compose multiple logical operators in transformation programs through concatenation.

Using logical operators, we can define the transformation learning problem as follows.

Definition 4.4.3. *Transformation Learning: Given a set of joinable row pairs $R = \{(R_s^i, R_t^i) | i \in [m]\}$ that can be viewed as input/output examples, and a predefined set of logical operations Θ, find a transformation $P = \theta_1 \cdot \theta_2 \cdot \theta_3 \cdot \ldots \theta_n$, $\theta_i \in \Theta$, such that*
(1) P is consistent with all examples in R, namely, $\forall i \in [m]$, $P(R_s^i)$ can produce the projection of R_t^i on some key columns K of T_t, denoted as $\Pi_K(R_t^i)$;
(2) P has minimum-complexity, measured as the number of logical operators used, among all other transformation programs that are consistent with examples in R.

This definition is in spirit consistent with principles such as Minimum Description Length [55] or Occam's razor [30] – if there are multiple candidate transformations that can explain all given examples, we use the simplest one and that is likely correct. For instance, the transformation in Example 4.3.1 requires 3 logical operators, and there exist no other programs with lower complexity.

The learning problem can be viewed as a search problem – each logical operator produces a partial output and has a unit cost. Like shortest path algorithms, we want to reach the goal state by producing the required output strings but with the least cost.

This motivates a program learning algorithm that searches for the best program by recursively expanding a partial transformation program using the logical operator that yields the most *progress*, which is defined as the total number of consecutive characters in the output examples produced by the partial transformation program.

Algorithm 4 Transformation learning by example.

```
Require: $R = \{I^i, O^i | i \in [k]\}$                         ▷ Input/output row pairs
 1: function TryLearnTransform($R = \{I^i, O^i | i \in [k]\}$)
 2:     while true do
 3:         $\theta \leftarrow$ FindNextBestLogicalOp($R$)
 4:         $P^i \leftarrow$ ExecuteOperator($\theta, I^i, O^i$), $\forall i \in [k]$
 5:         $O_l^i$ = LeftRemainder($O^i, P^i$), $\forall i \in [k]$
 6:         $\theta_l$ = TryLearnTransform($\{I^i, O_l^i | i \in [k]\}$)
 7:         if $\theta_l$ = null then
 8:             continue
 9:         end if
10:         $O_r^i$ = RightRemainder($O^i, P^i$), $\forall i \in [k]$
11:         $\theta_r$ = TryLearnTransform($\{I^i, O_r^i | i \in [k]\}$)
12:         if $\theta_r$ = null then
13:             continue
14:         end if
15:         $\theta$.left_child = $\theta_l$
16:         $\theta$.right_child = $\theta_r$
17:         return $\theta$                                          ▷ current root node
18:     end while
19: end function
```

This algorithm is outlined in Algorithm 4 and works as follows. For the given set of input/output examples, it finds the best logical operator θ (using FindNextBestLogicalOp) that produces the most progress towards the required output strings (which in this case is some key column of the output rows). The process of finding best logical operator involves iterating through all feasible parameters (e.g., split position, deliminator, select position etc.) given the input example. We execute the operator θ and extract partial output produced from the target output. We get what remains to the left and right in the target output, denoted as O_l^i and O_r^i, respectively. This produces two new instances of the problem with $\{I^i, O_l^i | i \in [k]\}$ and $\{I^i, O_r^i | i \in [k]\}$, which have the same structure as the original $\{I^i, O^i | i \in [k]\}$. So we recurse and invoke TryLearnTransform on the two smaller problems. The resulting operators, θ_l and θ_r, if learned successfully, are added as the left child and right child of θ, until all remaining target output have been consumed. If at certain level in the hierarchy TryLearnTransform fails to find a consistent transformation, we can backtrack by using the next best logical operator, or terminate if enough number of candidates have been tested.

In practice we impose a limit τ on the number of logical operators that can be used in a program P to bound the search space (e.g., $\tau = 16$). We found $\tau = 10$ to be sufficient to produce transformations needed to join all real scenarios we encountered. Setting a larger τ however has little impact on efficiency, because incorrect program generation paths are terminated quickly for failing to produce new operators consistent with the set of output examples.

We use the following example to illustrate this procedure.

Example 4.4.6. *From the example in Figure 4.1, suppose the first three rows from the right and left tables are given as learning examples for input/output row pairs, respectively. To learn transformation, suppose we use the first row* $\{[$`Obama, Barack(1961-)`$], [$`47.0`$]\}$ *as input and* $[$`Barack Obama`$]$ *as output, and we use the remaining two rows as validations. To generate the first logical operator for this row pair, we search over operators in* Θ *with all possible parameters (separators for* Split *up to a certain length, indexes for* Substring *that are valid for the given string, etc.), and pick the logical operator that yields the most progress. In this case it can be verified that the operator with the most progress is* Split-SplitSubstr, *which selects the first input element: [X =* `SelectK(input, 0)`*]; split by "(" and take the first element: [Y =* `SelectK(Split(X, "("), 0)`*]; split again by " " and take the second element: [Z =* `SelectK(Split(Y, " "), 1)`*]; take substring from position 1 to the end [1:-1]: [*`Substr(Z, 1, -1)`*]. This operator generates* `Barack` *for the first row,* `George W.` *for the second,* `Bill` *for the third, with a total gain of 19 characters (*$6+9+4$*), and an average gain of 49% for the required outputs across three rows (*$\frac{6}{12}$, $\frac{9}{14}$ *and* $\frac{4}{12}$*, respectively).*

With this first operator, the remaining required output strings to be covered are {" `Obama`*", "* `Bush`*", "* `Clinton`*" }. We again search for the logical operator that yields the most progress, for which we find* SplitSubstr *that splits by "*`,`*", takes the first element, and returns the full string. Now the remaining output strings are { " ", " ", " " }, which can be covered by adding a* Constant *operator that produces a space character. Finally, by concatenating these operators, we complete a candidate transformation program that can be tested on the full input tables.*

Through the following proposition, we show the success probability of transformation learning.

Proposition 4.4.3. *The learning procedure succeeds with high probability, if the transformation can be expressed using operators in* Θ*. The success probability is lower bounded by*

$$1 - \left(1 - \prod_{i \in [m]} \left(1 - \left(1 + (|S_I| + |S_I|^k)|S_O|^k\right) \frac{1}{|\Sigma|}^{k|S_i|}\right)\right)^T$$

where k is the number of independent examples used, T is the number of trials (each with k examples), $|S_I|$ and $|S_O|$ are the lengths of input/output examples, and $|S_i|$ is the length of the result of each intermediate step (for a logical operator).

A proof of this result can be found in Section 5.4 of the Appendix. This proposition shows that with T independent trials we can reduce the failure probability at a rate exponential in T, thus quickly improving success probability as T increases.

Ranking of Candidate Transformations.

Recall in Section 4.4.1, we generate groups of joinable row pairs, based on q-gram matches between each pair of columns C_s, C_t. For each such group, we select the top-k row pairs with the highest scores (typically 1-to-1 matches), and apply transformation learning for a fixed number of times, each of which on a random subset of the selected row pairs. We execute each learnt transformation on the original input tables, and pick the one that joins the most number of rows in the target table. By the definition of transformation join problem (Definition 4.3.1), the joining columns in the target table are key columns (1:1 or N:1 joins). A key-column join with high row coverage is likely to be meaningful in practice.

Algorithm 5 provides an overview for the transformation learning step. The LearnTransform(M) takes the q-gram matches M discovered in the previous step (Algorithm 3), which is grouped by the comlumn pairs. GenExampleSets generates multiple sets of examples using the groups of q-gram matches. In descending order of the average q-gram score, it goes through each group and generates a limited number of example sets. TopK takes at most k highest scored q-gram matches from each group. SubsetsOfSize creates r number of unique random subsets each with b number of q-gram matches. TryLearnTransform implements the learning algorithm, which is further expanded in Algorithm 4. ApplyTransform takes the transformation and applies it to the source table T_s. MaxBySCoverage returns the transformation (and its output) that results in the highest number of joined rows in the target table.

4.4.3 Join with Composite Key Columns

There exist cases where a key column in the target table is a *composite* column that joins with more than one columns in the source table. The right table in Figure 4.1 is such an example that has both the president names and their life spans. If we had used this composite column as the target, we would have failed to find any transformation, because the life span is missing from the left table.

While changing the direction of join in the example above would work, when both source and target key columns are composite, then neither direction would work. We use existing methods [15, 25] to split a composite columns by aligning substrings into multiple parts across all rows. For instance, one can split the key column in the right table in Figure 4.1 into three columns: the last name part before “,”, the first name part before “ (”, and the life span. Since composite columns usually have strong structural regularities (e.g., punctuations) this technique produces high quality splits in most cases, from which we can apply the Auto-Join again.

To summarize, we first attempt transformation joins in two directions. When neither direction results in a transformation, we can split the key columns of input tables before applying transformation join for the second time.

Algorithm 5 Overall steps for transformation learning

```
Require: $k$                                  ▷ Num. of row pairs to use in each group
Require: $r$                                  ▷ Num. of example sets from each group
Require: $L$                                  ▷ Max. num. of example sets to gen.
Require: $b$                                  ▷ Size of an example set
 1: function GenExampleSets($M$)
 2:     $sets \leftarrow \{\}$                                    ▷ Example sets
 3:     for all $(C_s, C_j), m \in M$ do                          ▷ Grouped by col. pair
 4:         $m \leftarrow$ TopK$(m, k)$
 5:         for all $set \in$ SubsetsOfSize$(m, b, r)$ do
 6:             $sets \leftarrow sets \cup \{(set, C_t)\}$
 7:             if $|sets| = L$ then
 8:                 return $sets$                              ▷ Reach max. num of sets
 9:             end if
10:         end for
11:     end for
12:     return $sets$
13: end function
14: function LearnTransform($M$)
15:     $p \leftarrow \{\}$                                       ▷ Learned transformations
16:     for all $set, C_t \in$ GenExampleSets$(M)$ do
17:         $t \leftarrow$ TryLearnTransform$(set)$
18:         if $t =$ null then
19:             continue                                        ▷ Failed to learn
20:         end if
21:         $C \leftarrow$ ApplyTransform$(t, T_s)$
22:         $p \leftarrow p \cup \{(C, C_t[y], t, |C \cap C_t|)\}$
23:     end for
24:     return MaxByCoverage$(p)$                               ▷ The best transform
25: end function
```

4.5 Scale Auto-Join to Large Tables

Auto-Join is used as a data exploration feature where interactivity is critical. In deploying this into a commercial data preparation system, a practical challenge we encountered is to efficiently scale the algorithm to tables with thousands or even millions of rows. In this section, we explain how to achieve such scalability for Auto-Join.

Given two tables T_s and T_t, each with millions of rows, with commodity hardware it is unlikely that we can achieve interactive speed if all values need to be indexed and probed. On the other hand, it would actually be wasteful to index/probe all values, because for the purpose of transformation learning we only need enough joinable row pairs for learning to be successful. Intuitively, we can sample rows from input tables, where the key is to use appropriate sampling rates to minimize processing costs but still guarantee success with high probability (the danger is that we may under-sample, thus missing join relationships that exist).

For Auto-Join, we use *independent row samples* from input tables, because unlike sampling techniques for equi-join or set-intersection where join keys are known and *co-ordinated sampling* can be used, in our problem, the join keys are not known *a priori*. Co-ordinated q-gram sampling is possible, but the cost of analyzing and hashing all q-grams is still prohibitive for large tables. Given these issues, we study a lightweight independent sampling for Auto-Join.

Let N_s, N_t be the number of rows in table T_s, T_t, and p_s, p_t be their sampling rates, respectively. Furthermore, let r be the *join participation rate*, defined as the fraction of records in the target table T_t that participate in the desired join. Note that the join participation rate needs to be reasonably high for sampling to succeed – if only one row in a million-row table participates in a join, then we need to sample a very large fraction of it to find the joinable row pair with high probability. In practice, r is likely to be high, because joins that humans find interesting likely involve a non-trivial fraction of rows. We conservatively set r to a low value (e.g., 1%), so that as long as the real join participation is higher we can succeed with high probability.

We formulate the problem of determining p_s and p_t as an optimization problem. The objective is to minimize the total number of sampled rows that need to be indexed and queried, which is $N_s p_s + N_t p_t$. The constraint is that we need to sample enough joinable row pairs with high probability. Since the join participation rate is r, at least $N_t p_t r$ rows from the target table T_t participate in join. Because each of these participating row sampled with probability p_s, leading to an expectation of $\mu = N_t \cdot p_t \cdot r \cdot p_s$ joinable row pairs in the sample. This can be seen as a Bernoulli process with a success probability of $p_t p_s r$ and N_t total trials.

As discussed in Section 4.4.2, we need a certain number of examples to constrain the space of feasible transformations enough to produce correct transformations. Let this required number be T (empirically 4 is enough in most cases). Let X be a random variable to denote the total number of joinable row pairs sampled. We want to find an upperbound for the probability that less than T joinable rows pairs are sampled, or $P(X \leq T)$.

Using the Multiplicative Chernoff Bound [14], we know X can be bounded by

$$P(X \leq (1-\delta)\mu) \leq e^{-\frac{\delta^2 \mu}{2}} \tag{4.5}$$

If we have $\mu \geq \frac{T}{1-\delta}$, we can upper-bound the failure probability as

$$P(X \leq T) \leq e^{-\frac{\delta^2 \mu}{2}} \tag{4.6}$$

For example, let $T = 4$, $\delta = 0.8$, we get $\mu = N_t p_t p_s r > \frac{T}{1-\delta} = 20$. Using Equation 4.6, we get $P(X \leq T) \leq e^{-\frac{0.64 \cdot 20}{2}} = e^{-6.4} < 0.0017$, or in other words, our success probability is at least 99.8%. So as long as we can ensure $\mu = N_t p_t p_s r > \frac{T}{1-\delta}$, then more than T joinable row pairs will be sampled with high probability. This becomes the constraint that completes our optimization problem:

$$\begin{aligned} \min \quad & N_s p_s + N_t p_t \\ \text{s.t.} \quad & N_t p_t p_s r \geq \frac{T}{1-\delta},\ p_t, p_s \in [0,1] \end{aligned} \tag{4.7}$$

Using Lagrange we obtain the following closed form optimal solution [8]:

$$p_t = \sqrt{\frac{T}{(1-\delta) r N_s}},\ p_s = \sqrt{\frac{T N_s}{(1-\delta) r N_t^2}} \tag{4.8}$$

The corresponding sample sizes can be written as $N_t p_t = \frac{N_t}{N_s}\sqrt{\frac{T N_s}{(1-\delta) r}}$ and $N_s p_s = \frac{N_s}{N_t}\sqrt{\frac{T N_s}{(1-\delta) r}}$, and both of them grow sub-linearly in N_s.

As a concrete example, suppose we have two tables both with 1M rows. For some fixed T and δ, such as $T = 4$ and $\delta = 0.8$ from the previous example that guarantees success with high probability, and $r = 0.1$, we can compute the sampling rates as $p_t = 0.014$ and $p_s = 0.014$, which translates to a small sample of 14K rows from the 1M-row tables. The optimized sampling significantly reduces processing costs and improves efficiency for Auto-Join on large tables.

4.6 Constrained Fuzzy Join

For datasets from the Web, inconsistencies are often found in how values are formatting and represented (e.g., with or without middle name and middle initials for names). Thus, an equi-join using transformation may miss some row pairs that should join, as Example 4.6.1 shows.

Example 4.6.1. *In Figure 4.2, the learned transformation that has the highest coverage on the target key column* `Email` *concatenates the first character of the first name with the last name in the left table, to get the email addresses. However, this transformation does not cover the target key in the third row* `mipayne@forsyth.k12.ga.us` *as it uses the first two characters in the first name. As a result, the third rows in the left and right tables cannot be equi-joined.*

Traditionally fuzzy join is used to deal with small value inconsistencies. However, given a wide space of parameters in fuzzy join such as the tokenization, distance function, and threshold, configuring a fuzzy join that works well for a given problem instance is difficult. This is particularly true for Auto-Join, as it is intended to be a data exploration feature in spreadsheets where users may not have no the expertise on fuzzy joins.

[8]When N_s and N_t are small, there may not be feasible solutions (the required p may be greater than 1). In such cases we use full tables.

We propose to automatically optimize a fuzzy join configuration using rows that are already equi-joinable as constraints, so that as we relax matching criteria in fuzzy join, we do not make these rows to join more than their equi-join results (which indicates that the corresponding fuzzy join is too lax). Although we use this optimization in the context of Auto-Join, the techniques here are of independent interest and can be used to optimize general fuzzy join.

Given a column C produced by transformations on the source table, and K a key column from the target table, where the join is 1:1 or N:1, our optimization objective is to maximize the number of rows in the target table that can be fuzzy-joined between C and K, subjecting to the constraint on join cardinality.

Specifically, given a tokenziation scheme t from a space of possible configurations (e.g., word, 2-gram, 3-gram, etc.), a distance function d (e.g., Jaccard, Cosine), and a distance threshold (normalized into a fixed range, e.g., [0,1]). The rows that can fuzzy join for some given t, d, s, denoted as $F_{t,d,s}(C, K)$, is defined as follow.

$$F_{t,d,s}(C, K) = \{v_k \mid \exists v_k \in K, v_c \in C, d_t(v_k, v_c) \leq s\} \tag{4.9}$$

where d_t is the distance d using a given tokenization t.

This objective alone tends to produce overly lax matches. The counteracting constraint is to respect join cardinality. Specifically, after using fuzzy join every value $v_c \in C$ cannot join with more than one value $v_k \in K$. This can be viewed as a key-foreign-key join constraint – a foreign-key value should not join with two key values (even with fuzzy join).

Additionally, we can optionally require that each $v_k \in K$ cannot join with more than one distinct $v_c \in C$. This is an optional constraint assuming that on the foreign key side, each entity is only represented with one distinct value. E.g., if we already have "`George W. Bush`" in a table, we would not have "`George Bush`" or "`George W. Bush Jr.`" for the same person. On the other hand a very close value "`George H. W. Bush`" in the same column likely corresponds to a different entity and should not join with the same key as "`George W. Bush`". This optional requirement helps to ensure high precision.

These requirements lead to the following two constraints in the optimization problem.

$$\begin{aligned} &\arg\max_{t,d,s} |F_{t,d,s}(C, K)| \\ &\quad s.t.\ |\{v_k \mid v_k \in K, d_t(v_c, v_k) \leq s\}| \leq 1, \forall v_c \in C \\ &\qquad |\{v_c \mid v_c \in C, d_t(v_c, v_k) \leq s\}| \leq 1, \forall v_k \in K \end{aligned} \tag{4.10}$$

We can search over possible t, d, and s from a given parameterization space. The following example illustrates how fuzzy join optimzation is used for joining tables in Figure 4.1.

Example 4.6.2. *Continue with Example 4.6.1, after applying the transformation, the output for `Missy Payne` in the left table is `mpayne@forsyth.k12.ga.us`, which cannot be equi-joined with the email `mipayne@forsyth.k12.ga.us` in the right table. Using 3-gram tokenizer and Jaccard distance, the distance between the two is 0.125. Thus, a distance threshold above 0.125 would join these two rows. On the other hand, if we use a larger distance threshold such as 0.4, `kmoore@forsyth.k12.ga.us` (transformation output from the left table) would join `mipayne@forsyth.k12.ga.us` (in the right table), breaking the second constraint in Equation 4.10 as `mipayne@forsyth.k12.ga.us` is joined with two distinct values.*

The fuzzy join optimization algorithm finds the maximum threshold that still satisfies the join constraints, thus the optimal threshold in this case is 0.2 *or* 0.3*.*

Due to the monotonicity of the objective function with respect to the distance threshold, we use binary search to find the optimal distance threshold.

Algorithm 6 provides the complete pseudo code for performing distance threshold optimization. OptimizeThreshold takes the derived source column from transformation C and the target key column K, and uses binary search to find the optimal distance threshold. CheckConstraint is called at each iteration of the search to verify if the current threshold satisfies the join constraints in Equation 4.10.

4.7 Experiments

In this section we discuss experimental results on join quality (precision/recall) as well as scalability.

4.7.1 Benchmark Datasets

Benchmarks. We constructed two benchmark, `Web` and `Enterprise`, using test cases from real datasets; as well as a synthetic benchmark `Synthetic`.

The `Web` benchmark is constructed using tables on the Web. We sampled table-intent queries from the Bing search engine (e.g., "list of US presidents"). We then used Google Tables[9] to find a list of tables for that query (e.g., U.S. presidents), and selected pairs of tables that use different representations but are still joinable under transformation (e.g., Figure 4.1). We searched 17 topics and collected 31 table pairs. We observe that tables on the Web often have minor inconsistencies (e.g., formatting differences, with or without middle initials in names, etc.) for the same entity mentions, which cannot be easily accounted for using transformations alone. This makes `Web` a difficult benchmark.

The `Enterprise` benchmark contains 38 test cases, each of which has a pair of tables extracted from spreadsheet files found in the intranet of a large enterprise (e.g., Figure 4.3 and Figure 4.4). The test cases are constructed by grouping tables with common topics. Comparing to `Web` that has mostly 1-to-1, entity-to-entity joins, `Enterprise` also has cases with hierarchical N:1 joins (e.g., Figure 4.3).

Lastly, since Warren and Tompa studied a close variant of the auto-join problem and used synthetic datasets for evaluation, as a validation test we reconstruct 4 datasets that they used [68] as the `Synthetic` benchmark. The 4 test cases, `UserID`, `Time`, `NameConcat`, and `Citeseer`, either split or merge columns to produce target tables [68].

In all these benchmark cases, equi-join would fail. We manually created the ground truth join result for each pair of tables, by determining what rows in one table should join with what rows from the other table.

Evaluation metrics. We use the following metrics to measure join quality. Denote by G the row pairs in the ground truth join results, J the joined row pairs produced by an algorithm. We measure join quality using the standard *precision* and *recall*, defined as:

$$precision = \frac{|G \cap J|}{|J|},\ recall = \frac{|G \cap J|}{|G|}$$

[9]`https://research.google.com/tables`

Algorithm 6 Find the distance threshold that produces the maximum fuzzy-join coverage on the target table (Equation 4.9) while satisfying the join constraints (Equation 4.10).

```
Require: $d_t$                                    ▷ Distance function using tokenization $t$
Require: $\delta \in (0.0, 1.0)$                                       ▷ Stopping condition
 1: function OptimizeThreshold($C$, $K$)
 2:     $coverage_{best} \leftarrow 0$
 3:     $a \leftarrow 0.0$, $b \leftarrow 1.0$
 4:     $t_0 \leftarrow 0.0, t \leftarrow a + (b - a)/2$
 5:     while $|s - s_0| > \delta$ do
 6:         if CheckConstraint($C, K, s$) then
 7:             $coverage \leftarrow F_{t,d,s}(C, K)$
 8:             if $coverage > coverage_{best}$ then
 9:                 $coverage_{best} \leftarrow coverage$
10:             end if
11:             $b \leftarrow s$
12:         else
13:             $a \leftarrow s$
14:         end if
15:         $s_0 \leftarrow s$
16:         $s \leftarrow a + (b - i)/2$
17:     end while
18:     if $coverage_{best} > 0$ then
19:         return $s_0$                                   ▷ Found optimal threshold
20:     end if
21:     return $-1.0$                                  ▷ No feasible threshold found
22: end function
23: function CheckConstraint($C$, $K$, $s$)
24:     $matched \leftarrow \{\}$
25:     for all $u \in$ Distinct($C$) do
26:         $n' \leftarrow 0$
27:         for all $v \in K$ do
28:             if $d_t(u, v) \leq s$ then
29:                 if $v \in matched$ then
30:                     return false
31:                 end if
32:                 $matched \leftarrow matched \cup \{v\}$
33:                 $n' \leftarrow n' + 1$
34:                 if $n' > 1$ then
35:                     return false
36:                 end if
37:             end if
38:         end for
39:     end for
40:     return true
41: end function
```

We also report the *F-score* that is the harmonic mean of precision and recall. When an algorithm produces empty join results, we do not include it in computing average precision, but we include it in average recall and F-score.

4.7.2 Methods Compared

We implemented 8 methods for comparison.

Substring Matching (`SM`). We implemented the algorithm by Warren and Tompa [68]. This algorithm uses a greedy strategy to find a *translation formula*, which is a sequence of indexes of the source columns' substrings that matches parts of the target column.

Fuzzy Join - Oracle (`FJ-O`). It is known that a major difficulty of using fuzzy join is the need to find the right configuration from a large space of parameters, which includes different tokenization schemes, distance functions, and distance thresholds, etc. To be more favorable to fuzzy join based methods, we consider an extensive combination of configurations. Specifically, for tokenization we use {Exact, Lower, Split, Word, and q-gram for $q \in [2, 10]$} (similar to ones used in [33]); for distance functions we consider {Intersect, Jaccard, Dice, MaxInclusion}; and for thresholds we use 10 equally-distanced values (e.g., {0.1, 0.2, ..., 1} for Jaccard). This creates a total of 520 unique parameter configurations. We execute each of these fuzzy joins on columns that are used in the ground truth as if they are known *a priori*, and we join each row with top-1 fuzzy match in the other table to maintain high precision. We report the best configuration that has the highest average F-score across all cases.

Note that this method acts much like an "Oracle" – it has access to not only the columns that join, but also the ground truth join result to "fine tune" its configuration for the best performance. These optimizations are not feasible in practice, so this provides an upper bound on what fuzzy join like methods can achieve.

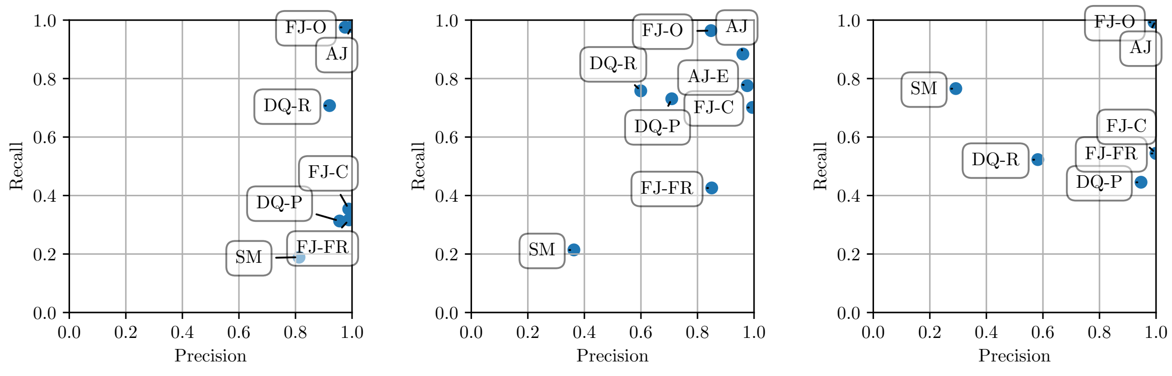
Fuzzy Join - Column (`FJ-C`). In this method, we perform fuzzy join on columns that participate in joins in the ground truth as if these are known, but without using detailed row-level ground truth of which rows should join with which for configuration optimization. We use techniques discussed in Section 4.6 to determine the best parameter configuration.

Fuzzy Join - Full Row (`FJ-FR`). This fuzzy join variant is similar to `FJ-C`, but we do not provide the information on which columns are used in join in the ground truth. As a result, this approach considers full rows in each table. This represents a realistic scenario of how fuzzy join would be used without ground truth.

Dynamic q-gram - Precision (`DQ-P`). Since q-grams already identify some joinable row-pairs (from which we generate transformations), one may wonder if it is sufficient to perform join using q-gram matches alone. In this method we use matches produced in Section 4.4.1, and only allow 1-to-1 q-gram matches to ensure high precision. Joinable row pairs are used directly as join result.

Dynamic q-gram - Recall (`DQ-R`). This algorithm is similar to `DQ-P`, except that we allow n-to-1 q-gram matches as join results. This produces results of higher recall but can also lead to lower precision compared to `DQ-P`.

Auto-Join (`AJ`). This is our Auto-Join algorithm. We create a variant **Auto-Join - Equality** (shorten as `AJ-E`) that only uses equality join without the fuzzy join described in Section 4.6.



(a) Enterprise Benchmark (b) Web Table Benchmark (c) Synthetic Benchmark

Figure 4.6: Average precision/recall comparison for all methods on three benchmarks.

4.7.3 Quality Comparison

In this section, we discuss the experimental results on the three benchmarks. Figure 4.6 shows the average precision and recall on all three benchmarks. Table 4.2, Figure 4.7 and 4.8 show the F-scores on all the benchmark datasets.

Enterprise Benchmark

`Enterprise` contains tables that are mostly clean and well structured – values are consistently encoded with few typos or inconsistencies as they are likely dumped from sources like relational databases (e.g., Figure 4.3 and Figure 4.4). Unlike other benchmarks, a significant fraction of joins are N:1 join through hierarchically relationships (e.g., Figure 4.3).

The precision and recall results are show in Figure 4.6a. First, Auto-Join (`AJ`) achieves near-perfect precision (0.9997) and recall (0.9781) on average. In comparison, the oracle baseline `FJ-O` has precision at 0.9756 and recall at 0.9755, which is inferior to Auto-Join. Recall that `FJ-O` is the Oracle version of fuzzy join that uses ground truth to find the best possible configuration, which provides an upper-bound for fuzzy join and is not feasible in practice. This demonstrates the advantage of transformation-based join over fuzzy join when consistent transformations exist. We note that in this test case using equality-join only `AJ-E` (with no optimized fuzzy join) produces virtually the same quality results, because values in this benchmark are clean and well structured.

Second, the `SM` algorithm achieves lower precision and recall then other baselines methods. This shows that their approach in building the translation formula using fixed substring indexes, as mentioned earlier in Section 4.7.2, is not expressive enough to handle transformations needed in real join scenarios we encountered.

Third, fuzzy join algorithms (`FJ-FR` and `FJ-C`) produced good precision due to our conservative fuzzy-join optimization. However, their recall is low, because in certain cases where the join values are hierarchical, non-joinable row pairs may also have low syntactic distance (e.g., Figure 4.3), which makes it difficult to differentiate between joinable and non-joinable row pairs using distance functions alone.

Lastly, `DQ-P` produces joins based on 1-to-1 q-gram matches, which has high precision but low recall. This is consistent with our analysis that 1-to-1 *q*-gram matches are often good joinable row pairs for transformation learning. On the other hand, `DQ-R` relaxes the matching constraints, and as expected

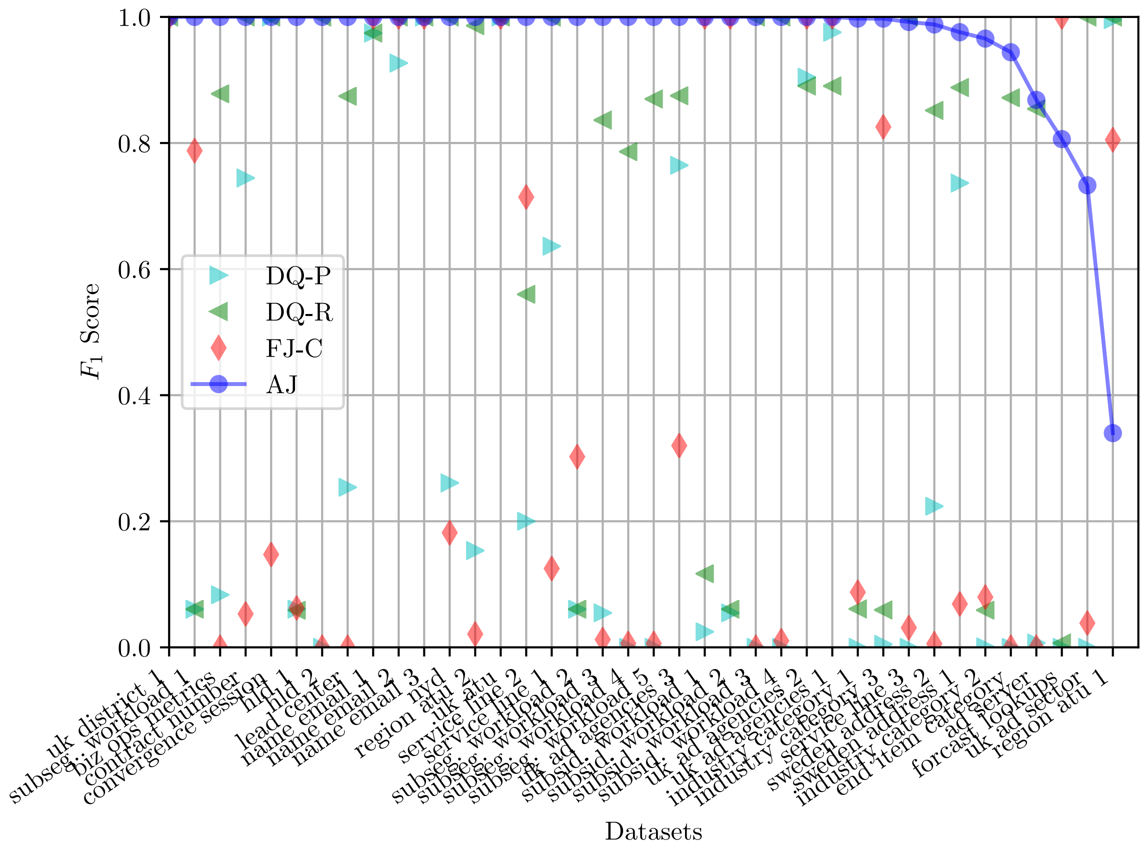


Figure 4.7: F-Scores on Enterprise Benchmark.

produces better recall but lower precision.

Figure 4.7 shows the F-scores on individual cases. It is clear that in most datasets, `AJ` achieved higher scores than the baselines, demonstrating that `AJ` is more resilient to complications such as N:1 joins and common substrings between joinable and non-joinable row pairs. In the test cases `uk ad sector` and `region atu 1`, `AJ` did worse than `DQ` methods. A close inspection reveals that it finds an alternative transformation that only covers a subset of the joinable results.

Bing Web Benchmark

Unlike `Enterprise` that have a significant number of N:1 joins, in `Web` most join cases are 1:1, entity-to-entity joins (e.g., Figure 4.1 and Figure 4.2), and are considerably more dirty with ad-hoc inconsistencies.

Figure 4.6b gives the quality comparisons. First, `AJ` has a considerably higher average precision than the oracle fuzzy join `FJ-O`, but a lower average recall. This is not surprising because `FJ-O` uses ground truth to optimizing its configuration parameters, which is not feasible in practice. We do notice that because `FJ-O` always joins a row with its top-1 match by score as long as the score is above a certain threshold, which leads to many false positives and thus lower precision. The problem is the most apparent for cases where most rows from one table do not actually participate in join. This is an inherent shortcoming of top-1 fuzzy join methods that `AJ` can overcome.

We see in Figure 4.6b that Auto-Join (`AJ`) has a higher average recall than its equality join version,

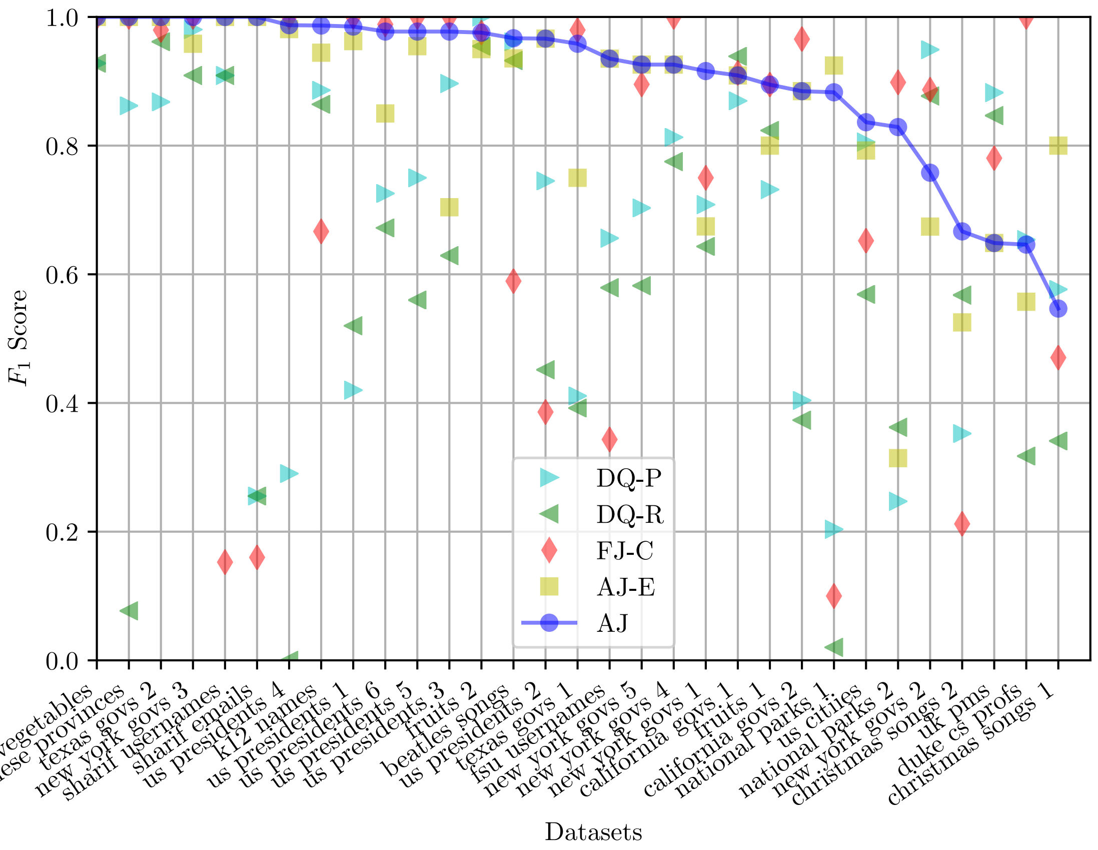


Figure 4.8: F-Scores on Web Table Benchmark.

`AJ-E` (0.8840 vs. 0.7757), but slightly lower precision (0.9504 vs. 0.9758). This is because inconsistencies exist in certain cases, where one correct transformation alone does apply to all rows that should join. In such cases, optimized fuzzy join brings significant gain in recall (≈ 0.11), with a small loss in precision (≈ 0.025).

`SM` does not perform well compared to other methods. Transformations required in `Web` benchmark are often too more complex for `SM` that relies on fixed substring indexes.

We analyze individual cases for which `FJ-C` produces higher F-scores than `AJ`, as shown in Figure 4.8. For cases like `uk pms`, we found that although `AJ` learnt the correct transformation and achieved a perfect precision, the fuzzy join step was not able to cover the rest of joinable row pairs that have inconsistencies in entities' naming. For complex cases like `duke cs profs`, the correct join actually requires more than one transformations in order to join all rows. Although `AJ` learns one transformation with perfect precision, it falls short in recall as not all joinable rows are covered. For these two datasets, the fuzzy distances between the non-joinable row pairs using the original join-columns are larger than when using the derived column. So it is easier for `FJ-C`, which uses the original join-columns, to differentiate between joinable and non-joinable row pairs and achieve higher F-score, even though `FJ-C` and `AJ` uses the same fuzzy join method.

Table 4.2: F-Scores on Synthetic Benchmark

	`Citeseer`	`NameConcat`	`Time`	`UserID`
DQ-P	0.9826	0.1264	0.0392	0.7572
DQ-R	0.9826	0.1356	0.2025	0.6638
FJ-C	0.4637	0.1651	1.0000	0.8795
SM	0.0291	0.1186	0.5464	0.7553
AJ	1.0000	1.0000	1.0000	1.0000

Synthetic Benchmark

`Synthetic` contains cases synthetically generated as described in prior work [68] using split or concatenation. The cases here are relatively simple and we use these as a validation test to complement with our previous benchmarks.

Figure 4.6c shows that `AJ` achieves perfect precision and recall, matching the oracle fuzzy join `FJ-O`. Other methods produce results similar to the previous benchmarks.

Table 4.2 shows the F-scores on individual cases. Both of `DQ-P` and `DQ-R` performs poorly on the `Time` dataset. `Time` is synthetically generated by concatenating three columns with second (0-59), minute (0-59), and hour (0-23) into a time column separated by ":". Due to the small space of numbers, there are many common substrings and few 1-to-1 or n-to-1 q-gram matches, thus the low scores of `DQ-P` and `DQ-R`. These two approaches work well on `Citeseer`, which has many 1-to-1 q-gram matches due to unique author names and publication titles. `AJ` achieved perfect F-scores on all datasets, since it just needs a few examples to produce the generalized transformations needed.

We found that `SM` achieved good recall in this benchmark, however, its average precision is relatively low (see Figure 4.6c). This result is not as well as what is reported in the original work [68]. This is likely because the method is data-sensitive, and it tends to fall into a local optimum with its use of greedy strategy in finding a translation formula. Since the translation formula is constructed incrementally by inspecting one source column at a time with no reverse-back, the addition of a single incorrect partial formula stops the whole algorithm from finding the globally optimal formula. This step is quite sensitive to the variance in the lengths of the substrings that matches with the target column. This is evident in Table 4.2, as `SM` did relatively better in `Time` and `UserID`, which has smaller variances (`Time` has zero variance, and `UserID` uses a fixed-length substring in the translation formula), while the scores in `Citeseer` and `NameConcat` are much lower.

4.7.4 Scalability Evaluation

We used the DBLP datasets[10] to evaluate the scalability of `SM`, `FJ-O`, `FJ-C`, and `AJ-E`. In the DBLP data set, each record has three fields: authors, title, and year of publication. For the purpose of scalability evaluation, we create a synthetic target table that is the concatenation of these three fields. We sample N records from the source table and the target table where N = {100, 1K, 10K, 100K, 1M}, and measure the corresponding end-to-end execution time. Some existing methods are very slow on large data sets so we set a timeout at 2 hours. Note that we omit results for `FJ-FR` since it is identical to `FJ-C` for

[10]`http://dblp.uni-trier.de/`

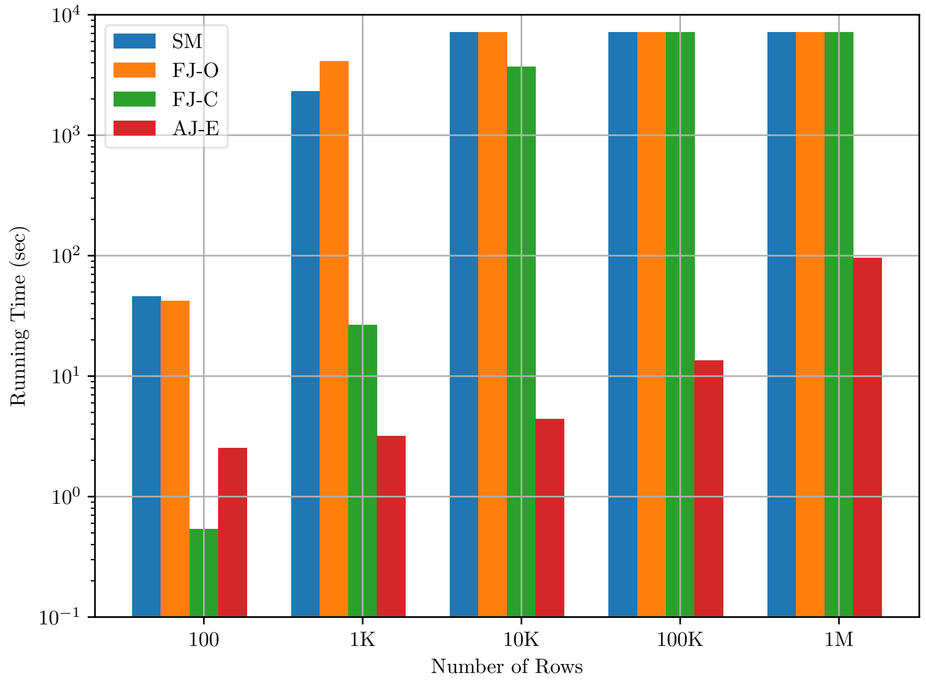


Figure 4.9: End-to-end running time comparison

this data set. We also do not compare with `DQ-R` and `DQ-P` since these are sub-components from the proposed `AJ` method.

Figure 4.9 shows the end-to-end running times. `AJ-E` is 2-3 orders of magnitude faster than existing methods. In particular, `SM` and `FJ-O` time out at 10K rows, and `FJ-C` times out at 100K rows; the runtime of these methods grow quickly with the table sizes.

For `AJ-E`, we break down the time into three stages – indexing, transformation-generation, and equi-join. We find that equi-join is efficient and accounts for less than 5% of overall execution time. The cost of transformation learning increases slowly, from 2.5 seconds at 100 rows to 6.4 seconds at 1M rows, as the number of attempts for transformations learning does not increase with the table size. The indexing time becomes a dominant factor as the number of rows grows over 100K.

In addition, we experimented Auto-Join without the optimized row sampling (Section 4.5) to see its impact on efficiency. Without sampling the algorithm reaches timeout on the data set with 1M rows, which shows the importance of sampling-based optimization for scaling to large data sets.

4.8 Summary

In this chapter, we have presented Auto-Join, an algorithm for automatically join tables by generating transformation programs, all without user input, to solve the problem of *ad hoc* script writing for data scientists. For efficiency, the algorithm uses a sampling technique to avoid using the whole table, and maintains interactive response time for tables under 10K rows. Auto-Join has been integrated into the

Azure Machine Learning Data Prep SDK[11]. Interesting future directions include automatically joining tables with semantic relationships, as well as complex domain-specific functions.

[11]`https://docs.microsoft.com/en-us/python/api/azureml-dataprep/azureml.dataprep.api.builders.joinbuilder`

Chapter 5

Conclusion

In this thesis, we introduced two important problems in data lakes: searching for joinable tables, and auto-generating syntactic transformation for joins. For the search problem, we presented two unique solutions: LSH Ensemble (Chapter 2) that leverages Locality Sensitive Hashing to find approximate results given a threshold on set containment, and JOSIE (Chapter 3) that finds exact top-k results through alternating probes of posting lists and sets guided by a cost model that minimizes read cost. For the join problem, we presented Auto-Join (Chapter 4) that generates syntactic transformations without user input, and scales to tables up to 10k rows while maintaining interactive speed. All the solutions have been validated over tables from real data lakes such as Open Data. The work presented in this thesis contributes to making it easier and faster for data scientists to find tables in data lakes, and perform *ad hoc* joins with transformations automatically – making data scientists more efficient in doing their job.

5.1 Other Contributions

This thesis mostly focuses on our algorithmic contributions. In this section, we will present two additional contributions toward data lake management: an interactive joinable table search system and an Open Source Python library for MinHash, LSH, and LSH Ensemble.

5.1.1 A Joinable Table Search System

We developed a user-facing system for interactive joinable table search. Figure 5.1 illustrates the overall design of this system, which won the Best Demo Award at VLDB 2017 [75]. The system works as follows. Given a data lake of tables, it first processes the tables to extract sets from columns, and then index these sets in a search index. From the user interface, the user inputs a table. The system then processes the table, extracts the query set, and then queries the search index. The search index returns a list of column and table IDs, using which the user interface renders the search results. The original system uses LSH Ensemble for the search index, however, it can easily be changed to use JOSIE.

The system is unique in that it allows the user to perform joins between the input table and any table found in the data lake, and apply a custom SQL query (with `SELECT`, `SUM`, `GROUP BY`, etc.) to transform the join result. This allows a user to quickly prepare and clean an extended table for analysis. Furthermore, the user can use the transformed table as the input table for another round of search, join,

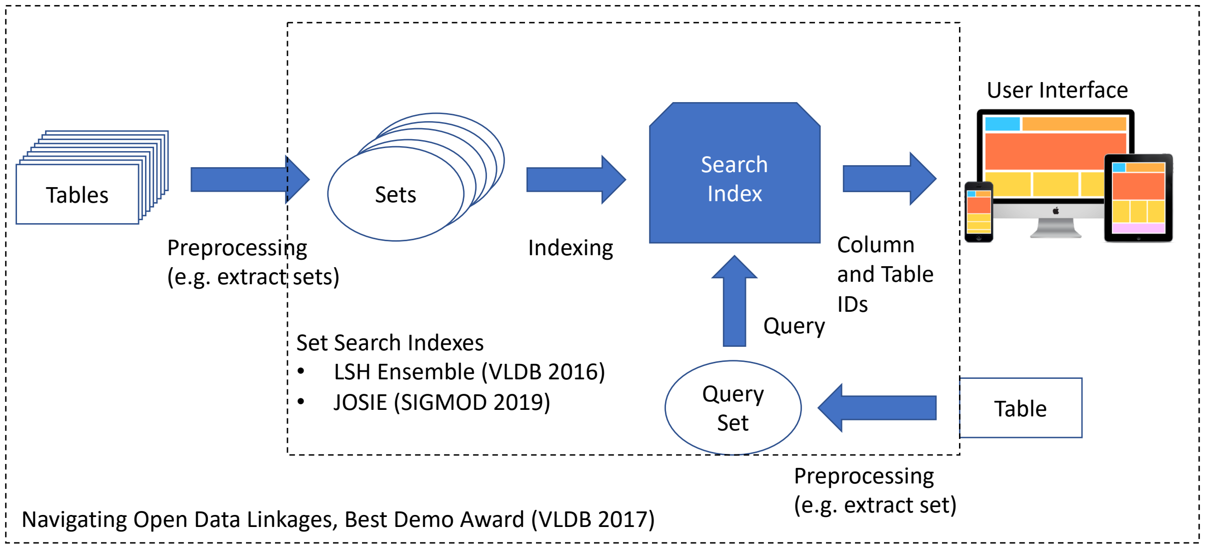


Figure 5.1: A Joinable Table Search System

and transformation. Thus, the user can expand the original input table with unlimited number of tables in the data lake.

Another feature is the sharing of any join or transformation result through a URL. This allows the user to share the current state of table preparation with others, who simply use the URL to recreate the same state. The system utilizes a result cache to avoid re-running the joins and transformations to recreate the state.

5.1.2 Open Source

During the development of LSH Ensemble, we implemented several data sketch algorithms including MinHash [9], MinHash LSH [31], MinHash LSH Forest [4], and our algorithm LSH Ensemble. The implementations are bundled in a Python package called `datasketch`, which can be installed via a `pip install datasketch` terminal command. The source code is open source, hosted on Github [1]. After making this Python package and source code available, we found there are many interested users. This discovery lead to an extensive effort to design a general API for all the algorithms, and write detailed documentation on their usage with code examples. The effort has since paid off: the number of users increased tremendously. To date, the `datasketch` Python package is a dependency of 85 Python projects including ones from Google and Stanford NLP Lab. The source code repository has merged 29 pull requests containing code contributions from 9 external developers, closed 51 issues related to usage questions and bugs, and accumulated over 830 stars (i.e., endorsements). This Open Source project has made it much easier for developers to integrate the products of our research within their own systems, and encouraged subsequent research work. For example, the Aurum Data Discovery System [26] uses MinHash LSH from `datasketch`.

[1]`https://github.com/ekzhu/datasketch`

5.2 Lessons Learned

We have learned some valuable lessons through all the hard work put into this thesis. These lessons include insights and approaches to algorithm design, real data distributions, performance tuning, and Open Source strategy. The lessons are not meant to be universal rules that guarantee positive gain, rather, they are useful case studies to be reviewed when approaching a new problem.

5.2.1 The Two-Stage Design Principle of Retrieval Processes

To design a retrieval process, such as searching for joinable tables in a data lake, one can often apply the two-stage design principle: first *candidate generation* and then *candidate refinement*.

In the candidate generation stage, the retrieval algorithm identifies, typically using a search index, a possibly large set of candidates that may be qualified for showing the end user. This stage is intended for maximizing recall, so false positive candidates are tolerated. LSH Ensemble is a good example of a search index that is designed for candidate generation: given a containment threshold, it finds candidate sets that likely meet the threshold (with high probability), while also including false positives in the result. The candidate generation process can impact the overall search performance by producing too many false positive candidates, so it is important to tune for better precision without sacrificing recall. LSH Ensemble solves this problem by 1) using the approximate containment to Jaccard similarity translation function that does not introduce false negatives, and 2) using partitioning to reduce false positives.

In the candidate refinement stage, the retrieval algorithm uses a more expensive approach, typically a linear scan, to compute the relevance scores of the candidates generated in the previous stage. This stage is intended for maximizing precision, so false positives are typically removed. For candidate sets generated by LSH Ensemble, we simply compute their exact containment, and remove sets that do not satisfy the containment threshold.

The two-stage design principle is beneficial when the candidate generation stage has high recall, especially when there is a fast search index (e.g., LSH Ensemble or JOSIE) that can produce candidates based on a lower bound or "prerequisite" of the actual relevance measure. For example, in joinable table search, the "prerequisite" can be a containment threshold, and the actual relevance measure can be a combination of different syntactic and semantic signals such as join columns' data type (e.g., date, geographic location, person names, etc.), number of rows in the join result, and whether the join columns are key columns. The refinement stage can also include additional steps such as filtering out semantically irrelevant candidate columns that do not lead to meaningful join (e.g., joining years with integer keys). Such complex relevance measures may not be supported by any available search index, so the candidate generation stage is used to speed up the overall retrieval process without resorting to a linear scan of all the data being searched over.

5.2.2 Evaluate Using Real Data

Algorithms' performance can be significantly affected by specific data characteristics such as set size and frequency distributions. Thus, when evaluating an algorithm, it is important to go beyond synthetic benchmarks and use real data because real data may have characteristics that the synthetic benchmarks do not capture.

During the process of developing LSH Ensemble, we used a synthetic benchmark generated using a normal distribution of set sizes. This led to a very different algorithm, which performed extremely well on the benchmark, but poorly on the Open Data lakes. It took some time before we realized the difference in set sizes was the root cause. After an extensive profiling of the Open Data lakes and experimenting with different algorithms, we understood that the set size distribution is also the reason why Asymmetric Minwise Hashing [60] has high accuracy on Twitter data but poor accuracy on data lakes. In the end, we looked for ways to reduce the difference in set sizes, and that led to the discovery of the partitioning approach used by LSH Ensemble.

JOSIE was developed with real data lakes in mind. When evaluating the state-of-the-art exact top-k techniques (`MergeList` and `ProbeSet`) for overlap set similarity search, we found that `MergeList` outperformed `ProbeSet` on the Open Data lakes, but `ProbeSet` outperformed `MergeList` on WDC Web Tables. We immediately knew this was caused by specific data characteristics. Upon investigation, we found that the Open Data lakes have large sets and relatively short posting lists, while WDC Web Tables have small sets and relatively long posting lists. `ProbeSet` mostly reads sets, so it performs well on WDC Web Tables, and `MergeList` only reads posting lists, so it performs well on the Open Data lakes. This discovery lead to the development of JOSIE, which always picks the direction that yields lower read costs. In the end, JOSIE outperforms both `MergeList` and `ProbeSet`.

5.2.3 Use Simple Methods to Optimize for Performance

This lesson is in spirit similar to the classic System-R approach to query optimization, which only considers "left-deep" query plans in order to shrink the plan space and search complexity. The rational is that the query plan optimization should be inexpensive and much cheaper than executing the query plan itself. JOSIE uses a cost model to optimize the probing sequence for minimal read cost, and the cost model requires an estimation of the expected intersection size between the query set and any candidate set. When developing JOSIE, we have considered different methods for intersection size estimation: some methods rely on global co-occurrence probabilities of token pairs, and one method uses the Hypergeometric probability distribution to estimate the total number of overlapping tokens. In the end, we settled with our initial method that uses a uniform probability distribution assumption of overlapping tokens in the query set (see Section 3.3.2), because this method is the fastest to execute, requires no expensive computation of prior distributions (e.g., global co-occurrence probabilities), and yields a large improvement in overall query performance at little expense.

5.2.4 Rare Signals are Strong Signals

This lesson comes from the development of Auto-Join. In searching for correct input/output learning example row pairs in Auto-Join, the space of all possible learning examples are all pairs of rows from the source and target tables – most of pairs are incorrect. To efficiently locate good learning examples, Auto-Join starts with row pairs with rare q-grams that occur exactly once in the source table and once in the target table. The rational behind this is that it is very unlikely for such q-gram to occur, and when they do, they provide very strong signals for the corresponding row pairs to have potential join relationships. This rational led to Auto-Join's first component that discovers a ranked list of input/output examples for transformation learning. In general, when searching over a large space for solutions, we may want to start with potential solutions with rare feature instances.

5.3 Next Step: A Self-Service Data Discovery System

An important next step from this thesis is to combine all three techniques: LSH Ensemble, JOSIE, and Auto-Join, together with other components based on very recent new data discovery work developed in parallel with our work [51, 26, 35, 27], to create a self-service data discovery system for data scientists. The objective of such a system is to provide table discovery over multiple massive data lakes such as Open Data and enterprise data lakes, to help data scientists quickly identify relevant data sets and augment (via join or union) their existing tables.

5.3.1 Generalized Joinable Table Search

One important feature of this system is generalized joinable table search – given a query table, the system finds tables that can be joined with the query table by equi-join or transformation-based join. The approximate (i.e., LSH Ensemble) and exact technique (i.e., JOSIE) can be used together: the system can issue both the approximate and exact queries at the same time, and present the results from the approximate algorithm as soon as they are available, while the exact results can be used to refine the final output. Since the most promising results are likely to be first returned by the approximate algorithm and the user needs at least a few seconds to read each result, the less promising results will be refined by the time the user reaches them.

A more challenging problem is how to find joinable tables with transformation-based joins. The search algorithms, LSH Ensemble and JOSIE, presented in this thesis assume a simple model for tables that represents each column as a set, and joinable values must match exactly (i.e., equi-join). On the other hand, the join algorithm, Auto-Join, solves the problem when the joinable columns do not match exactly, but require syntactic transformations. One simple approach to make columns transformation-join searchable is to tokenize data values based on a syntactic pattern that is present in this column, and index the column using tokens (or a MinHash of the tokens). For example, data values in an `Email` column with the syntactic pattern “`first.last@host.com`” may be tokenized into “`first`”, “`last`”, and “`host.com`”, and the resulting tokens are inserted into a search index as keys to this column. The rational behind this approach is that transformations typically do not break tokens, so two joinable columns should have many tokens in common. A regular expression synthesizer such as the one developed by Ilyas et al. [35] can be used to find the syntactic pattern. Furthermore, we can assign higher weights to the more infrequent tokens that match with the query column, because the infrequent token-matches are likely the result of some transformations, as described in Section 4.4.1 of this thesis.

Handling transformation-based join improves the recall, however, it is equally important to improve the precision of the search result. We can make two additions to the existing search algorithms for better precision. The first addition is to filter out irrelevant tables that are joinable by pure accidental value matches. For example, one may find tables with an integer key column that is joinable with a year column in the query table. To filter out these irrelevant tables, we can use the word embedding of column names when available (used by Fernandez et al. [27]) to compute the semantic similarity between the query column and joinable columns in the candidate tables, and then remove candidate tables that only have joinable columns that are semantically dissimilar with the query column. The second addition for better precision is to utilize multi-column join signals – rank candidate tables that are joinable on more than one column with the query table higher than those that are joinable on one column only. The rational behind this is that multi-column joins are likely to be more meaningful than single column joins. For example, a table T_1 that joins on both `Province` and `City` columns is likely a better candidate than a table T_2 that joins only on `Province`, because the records in T_1 share more information with the records in the query table.

5.3.2 Unionable Table Search

Another important feature of a self-service data discovery system is unionable table search. Different from join, the union operation puts records from different tables into a common schema. For example, we may want to create a master rental listing by taking records obtained from different real estate agencies. The unionable table search problem, thus, is to find tables that can be unioned with the query table, so the user can create a "master list" similar to the one in the previous example. Unionable table search can be useful to data scientists who are looking for more training data to train a machine learning model. In collaboration with Fatemeh Nargesian, we have developed a unionable table search algorithm [51] that scales to data lakes with hundreds of thousands of tables. This algorithm utilizes an ensemble of column-based signals (syntactic, ontological, and natural language) to identify columns semantically similar with the query table's columns, and then uses a scoring function that takes the column-based signals to produce a ranked list of candidate tables that are unionable with the query table. Another approach to find unionable tables using schema information has been developed by Lehmberg et al. [39], and it has been tested on WDC Web Tables. We can use these algorithms to implement a unionable table search feature of the data discovery system. Ultimately, a self-service data discovery system should be able to determine if a candidate table should be unioned or joined with a query table (not making the data scientist choose *a priori*).

5.4 An Outlook on Data Lake Management[2]

Our vision for the future centers on going from database management to data lake management. A data lake is a massive collection of raw files that: (1) may be hosted in different, typically distributed, storage systems; (2) may vary in their formats; (3) may not be accompanied by any useful metadata or may use different formats to describe their metadata; and (4) may change autonomously over time. Enterprises have embraced data lakes for a variety of reasons. First, data lakes decouple data producers (for example, operational systems) from data consumers (such as, reporting and predictive analytic systems). For data science, data lakes provide a convenient storage layer for experimental data, both the input and output of data analysis and learning tasks. The creation and use of data can be done autonomously without coordination with other programs or analysts.

Figure 5.2 shows a very high-level view of the architecture we envision for a data lake. The data sources may include legacy operational systems (operating in Cobol or other older formats), information scrapped from the Web and social media (including the Open Data and Web Tables we have studied), or data from for-profit data brokers (such as Thompson Reuters or LexisNexis). The actual type information and metadata may be represented in numerous different formats. Other data may be pure documents (unstructured data) or semi-structured logs, or social media information.

My thesis has contributed two of the first indices for data discovery over tables (or sets) in a data lake. Going forward, we envision a growing interest in coupling information extraction with new forms

[2]This section will be published in "Data Lake Management: Challenges and Opportunities" [50], a collaboration with Fatemeh Nargesian, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena.

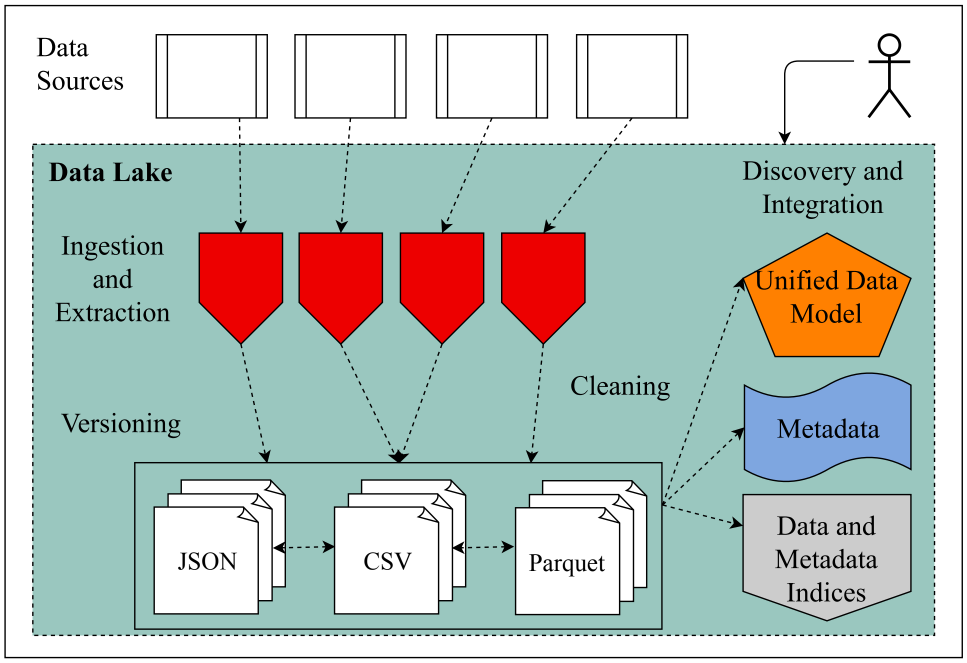


Figure 5.2: The Architecture of a Data Lake Management System [50]

of indices that facilitate data discovery using metadata and semantics.

Current data lakes provide reliable storage for datasets together with a computational framework (such as Hadoop or Spark), along with a suite of tools for doing data governance (such as access control), data discovery, extraction, cleaning, and integration. These tools help individual teams to use (and create) data in a data lake. But many challenges remain. First, we are only at the beginning of being able to exploit the work of others (the search, extraction, cleaning and integration effort of others) to help in new uses of a data lake. How can an analysis or search done by one data scientist improve the discovery or analysis of tables by another? Second, data lakes are currently mostly intermediate repositories for data. Currently, this data does not become actionable until it is cleaned and integrated into a traditional DBMS or warehouse. A grand challenge for data lake management systems is to support on-demand query answering – meaning data discovery, extraction, cleaning, and integration done at query time over massive collection of files that may have unknown structure, content, and data quality. Only then would the data in data lakes become actionable.

Appendix

Proof of Proposition 2.4.2

Recall from Section 2.4.5, $N_{|X|}$ is the number of sets with size x in the interval $[l, u]$. The expected number of false positives in the interval produced by using the effective containment thresholds is given by:

$$N_{l,u}^{\text{FP}} = \sum_{|X| \in [l,u]} N_{|X|} \cdot \Pr[X \textit{ is } \text{FP}]$$

where $\Pr[X \textit{ is } \text{FP}]$ is the probability of a set becoming a false positive given its size is $|X|$. The effective containment threshold for a set with size $|X|$ is given in Proposition 1 as:

$$t_{|X|} = \frac{(|X| + |Q|)t^*}{u + |Q|}$$

where $|Q|$ is the query set size, and t^* is the query-given containment threshold.

For any set with size $|X|$, its containment is bounded by $\min\{1.0, |X|/|Q|\}$. Thus, given different $|Q|$, the maximum and minimum containment from sets in the interval can be different, and the probability of false positive can be different as well. We separate the proof into 5 cases.

Case 1: $t^*|Q| \leq l$

Since the lower bound of the interval is greater than minimum set size required to satisfy the containment threshold, all sets in the interval have non-zero probabilities of becoming false positives. The false positive probability is given as

$$\begin{aligned} \Pr[\text{FP} \,|\, |X| \geq t^*|Q|] &= (t^* - t_{|X|})/t^* \\ &= 1 - \frac{|X| + |Q|}{u + |Q|} \end{aligned}$$

We can evaluate $N_{l,u}^{\text{FP}}$ as follows:

$$\begin{aligned} N_{l,u}^{\text{FP}} &= \sum_{|X| \in [l,u]} N_{|X|} \cdot \frac{t^* - t_{|X|}}{t^*} \\ &= \sum_{|X| \in [l,u]} N_{|X|} \cdot \left(1 - \frac{|X| + |Q|}{u + |Q|}\right) \leq \sum_{|X| \in [l,u]} N_{|X|} \cdot \left(1 - \frac{|X|}{u}\right) \end{aligned}$$

Case 2: $t_l|Q| \leq l < t^*|Q|$ **and** $t^*|Q| \leq u$

For sets in the sub-interval $[l, t^*|Q|)$, their maximum possible containment do not satisfy the query-given containment threshold, but satisfies the effective containment threshold. The false positive probability of these sets becomes

$$\begin{aligned}\Pr[\text{FP}|t_l q \leq x < t^*|Q|] &= (|X|/|Q| - t_{|X|})/(|X|/|Q|) \\ &= 1 - t_{|X|}\frac{|Q|}{|X|} \\ &= 1 - \frac{|X|+|Q|}{u+|Q|} \cdot \frac{t^*|Q|}{|X|}\end{aligned}$$

$N^{\text{FP}}_{l,u}$ now can be evaluated as

$$\begin{aligned}N^{\text{FP}}_{l,u} &= \sum_{|X|\in[l,t^*|Q|)} N_{|X|} \cdot \Pr\left[FP|t_l|Q| \leq |X| < t^*|Q|\right] + \sum_{|X|\in[t^*|Q|,u]} N_{|X|} \cdot \Pr[\text{FP}||X| \geq t^*|Q|] \\ &\leq \sum_{|X|\in[l,u]} N_{|X|} \cdot \Pr[\text{FP}||X| \geq t^*|Q|]\end{aligned}$$

Since $|X| < t^*|Q|$ in the sub-interval $[l, t^*|Q|)$, $\Pr\left[\text{FP} \mid t_l|Q| \leq |X| < t^*|X|\right] \leq \Pr[\text{FP}||X| \geq t^*|Q|]$. We still have the inequality:

$$N^{\text{FP}}_{l,u} \leq \sum_{|X|\in[l,u]} N_{|X|} \cdot \left(1 - \frac{|X|}{u}\right)$$

Case 3: $l < t_l|Q|$ **and** $t^*|Q| \leq u$

Now the maximum possible containment for sets in the sub-interval $[l, t_l|Q|)$ are below the effective containment threshold. For these sets, the false positive probability is zero

$$\Pr[\text{FP}||X| < t_l|Q|] = 0$$

Following the same steps as before, we can show the inequality still holds.

$$\begin{aligned}N^{\text{FP}}_{l,u} &= \sum_{|X|\in[l,t_l|Q|)} N_{|X|} \cdot \Pr[\text{FP}||X| < t_l|Q|] + \sum_{|X|\in[t_l|Q|,t^*|Q|)} N_{|X|} \cdot \Pr[\text{FP}|t_l|Q| \leq |X| < t^*|Q|] \\ &+ \sum_{|X|\in[t^*|Q|,u]} N_{|X|} \cdot \Pr[\text{FP}||X| \geq t^*|Q|] \\ &\leq \sum_{|X|\in[l,u]} N_{|X|} \cdot \Pr[\text{FP}||X| \geq t^*|Q|] \\ &\leq \sum_{|X|\in[l,u]} N_{|X|} \cdot \left(1 - \frac{|X|}{u}\right)\end{aligned}$$

Case 4: $l < t_l|Q|$ **and** $t_u|Q| \leq u < t^*|Q|$

Since the probabilities of false positives in the intervals $[l, t_u|Q|)$ and $[t_u|Q|, u)$ are both less than $\Pr[\text{FP}||X| \geq t^*|Q|]$, the inequality still holds, follow the same reasoning.

Case 5: $u < t_u|Q|$

The probability of false positive is 0 in the complete interval $[l, u]$, thus the inequality still holds.

Thus Proposition 2. □

Proof of Proposition 4.4.3

Recall in Section 4.4.2, Proposition 4.4.3 states that if a transformation required is expressible in the language using operators in Θ and examples selected for learning are independent of each other, then the learning process has a high probability of success.

To show that this is the case, we start by analyzing a simplified scenario with only two operators, Substr and Concat. Given an input string of length S_I, and an output string of length S_O that are selected as an example pair. Since only Substr and Concat are used in generating the transformation, the desired transformation in this language is bound to be a concatenation of m substrings, where m is the size of the transformation (or the number of operators).

Our algorithm greedily selects the best operator in each step, which is the one with the most gain towards the target output. Let $\{S_1, S_2, ...S_m\}$ be the sequence of substrings generated, whose concatenation produces S_O. We now compute the probability that one substring segment S_i is correctly generated. In each step i, the reason that it may fail is because there exists another q-gram in S_I with a different match in S_O, whose length is at least $|S_i|$. The probability that this event happens for one example can be bounded by $|S_I||S_O|\frac{1}{|\Sigma|}^{|S_i|}$[3]. For k number of independent examples, denote by $P_k(str)$ the probability of confusing a true Substr operator with a different Substr, $P_k(str) = |S_I|(|S_O|\frac{1}{|\Sigma|}^{|S_i|})^k$, which decreases exponentially in k. Note that $|S_I|$ is not raised to the power of k because all k examples are required to have the substring at the same position, but for S_O there is no restriction on starting positions. As a result the success probability for the i-th step is $1 - |S_I|(|S_O|\frac{1}{|\Sigma|}^{|S_i|})^k$, and the success probability for a given set of k examples across m steps is bounded by $\prod_{i\in[m]}\left(1 - |S_I|(|S_O|\frac{1}{|\Sigma|}^{|S_i|})^k\right)$. Given that we try a fixed T number of random example-sets (e.g. $T = 128$), and the overall transformation succeeds as long as one such example set can produce the correct transformation, so the overall success probability is

$$1 - \left(1 - \prod_{i\in[m]}(1 - P_k(str))\right)^T$$

We now consider the scenario with operator Const in addition to Substr and Concat. At step i, we may produce an incorrect Const operator, in place of a correct Substr, or a correct but different Const. Let $P_k(const)$ be the probability. $P_k(const) = |S_O|(\frac{1}{|\Sigma|}^{|S_i|})^k$, because it also fails when there exists a constant q-gram in S_O whose length is at least $|S_i|$.

Note that $P_k(str)$ now indicates the probability of confusing a Substr with Const and a different Substr, which is the same as the value computed above, because the only case where it is produced incorrectly is when the substring length exceeds that of the correct program $|S_i|$.

[3]Characters are assumed to be sampled uniformly at random from Σ for simplicity of this analysis. A different character-level distribution model can be plugged here in place of $\frac{1}{|\Sigma|}^{|S_i|}$ to produce results with the same structure and similar overall results.

$$1 - \left(1 - \prod_{i \in [m]} (1 - P_k(str) - P_k(const))\right)^T$$

We now describe the scenario with the addition of operators that use Split (SplitSubstr and SplitSplitSubstr). Split can be viewed as modifying the relative positions of each input string, but does not alter local q-grams in other ways. So in the best case it can shift the starting position of q-grams. The probability of mistakenly producing a Split instead of others is thus $P_k(split) = (|S_I||S_O|\frac{1}{|\Sigma|}^{|S_i|})^k$. Note that compared to $P_k(str)$ here $|S_I|$ is raised to the power of k, because starting position of input string is no longer constrained to be the same.

Combining, the success probability is lower bounded by

$$1 - \left(1 - \prod_{i \in [m]} (1 - P_k(str) - P_k(const) - P_k(split))\right)^T$$

which can be rewritten as

$$1 - \left(1 - \prod_{i \in [m]} \left(1 - \left(1 + (|S_I| + |S_I|^k)|S_O|^k\right) \frac{1}{|\Sigma|}^{k|S_i|}\right)\right)^T$$

The failure probability becomes exponentially small with more number of attempts T. The failure probability generally decreases when using more number of examples k, but increases with m that indicates more complex programs.

For illustration, we plug in numbers for concreteness. The program generated in Example 4.4.6 for tables in Figure 4.1 has the following parameters: $|S_I| = 29$, $|S_O| = 19 + 4 = 17$ (we pick the longest input/output, in this case the fourth row to bound the probability). Value of $m = 3$ since the desired program has 3 logical operators, and $|S_1| = 12, |S_2| = 4, |S_3| = 1$. Assume we use 3 examples to generate programs so $k = 3$, and $|\Sigma| = 52$. Even with $T = 1$ the success probability is ≈ 0.999 [4]. When failure probability of individual trials is more significant, with T repeated independent trials we can quickly reduce the failure probability at exponential rate, and thus produce a high overall success rate.

[4] internally when multiple operators can produce the same output sequence with the same score, Const will be picked over other operators for the simplicity of its explanation. Thus the last operator does not have a confusion probability and will succeed with probability of 1.

Bibliography

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. *PVLDB*, pages 918–929, 2006.

[2] David Aumueller, Hong Hai Do, Sabine Massmann, and Erhard Rahm. Schema and ontology matching with COMA++. In *SIGMOD*, pages 906–908, 2005.

[3] Jana Bauckmann, Ulf Leser, Felix Naumann, and Veronique Tietz. Efficiently detecting inclusion dependencies. In *ICDE*, pages 1448–1450, 2007.

[4] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH forest: Self-tuning indexes for similarity search. In *World Wide Web Conference*, pages 651–660, 2005.

[5] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[6] Alexander Behm, Chen Li, and Michael J. Carey. Answering approximate string queries on large data sets using external memory. In *ICDE*, pages 888–899, 2011.

[7] Christian Biemann. *Structure Discovery in Natural Language*. Theory and Applications of Natural Language Processing. Springer, 2012.

[8] Alexander Bilke and Felix Naumann. Schema matching using duplicates. In *ICDE*, pages 69–80, 2005.

[9] A. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, pages 21–28, 1997.

[10] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Computer Networks and ISDN Systems*, 1997.

[11] Michael J. Cafarella, Alon Y. Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. Webtables: exploring the power of tables on the web. *PVLDB*, 1(1):538–549, 2008.

[12] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.

[13] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[14] H. Chernoff. A note on an inequality involving the normal distribution. *Annals of Probability*, 1981.

[15] Xu Chu, Yeye He, Kaushik Chakrabarti, and Kris Ganjam. Tegra: Table extraction by global record alignment. In *Proceedings of SIGMOD*, 2015.

[16] Joel Coffman and Alfred C Weaver. An empirical performance evaluation of relational keyword search techniques. In *TKDE*, pages 30–42, 2014.

[17] Edith Cohen and Haim Kaplan. Summarizing data using bottom-k sketches. In *PODC*, pages 225–234, 2007.

[18] CrowdFlower. 2017 data scientist report. `https://visit.crowdflower.com/WC-2017-Data-Science-Report_LP.html`, 2017.

[19] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*, pages 253–262, 2004.

[20] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. The data civilizer system. In *CIDR*, 2017.

[21] Dong Deng, Albert Kim, Samuel Madden, and Michael Stonebraker. Silkmoth: An efficient method for finding related sets with maximum matching constraints. *PVLDB*, 10(10):1082–1093, 2017.

[22] Dong Deng, Guoliang Li, Jianhua Feng, and Wen-Syan Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013.

[23] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang, and Jianhua Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, pages 340–351, 2014.

[24] Songyun Duan, Achille Fokoue, Oktie Hassanzadeh, Anastasios Kementsietsidis, Kavitha Srinivas, and Michael J. Ward. Instance-based matching of large ontologies using locality-sensitive hashing. In *ISWC*, pages 49–64, 2012.

[25] Hazem Elmeleegy, Jayant Madhavan, and Alon Halevy. Harvesting relational tables from lists on the web. *The VLDB Journal*, 20(2):209–226, April 2011.

[26] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. Aurum: A data discovery system. In *ICDE*, pages 1001–1012, 2018.

[27] Raul Castro Fernandez, Essam Mansour, Abdulhakim Ali Qahtan, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, Michael Stonebraker, and Nan Tang. Seeping semantics: Linking datasets using word embeddings for data discovery. In *ICDE*, pages 989–1000, 2018.

[28] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. Set similarity joins on mapreduce: An experimental survey. *PVLDB*, 11(10):1110–1122, 2018.

[29] Niloy Ganguly, Andreas Deutsch, and Animesh Mukherjee. Dynamics on and of complex networks: Applications to biology, computer science, and the social sciences. 2009.

[30] Hugh G. Gauch. *Scientific Method in Practice.* Cambridge University Press, 2003.

[31] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[32] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *SIGPLAN*, 2011.

[33] Oktie Hassanzadeh, Ken Q. Pu, Soheil Hassas Yeganeh, Renée J. Miller, Lucian Popa, Mauricio A. Hernández, and Howard Ho. Discovering linkage points over web data. *PVLDB*, 6(6):444–456, 2013.

[34] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[35] Andrew Ilyas, Joana M. F. da Trindade, Raul Castro Fernandez, and Samuel Madden. Extracting syntactical patterns from databases. In *ICDE*, pages 41–52, 2018.

[36] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[37] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *SIGMOD*, 2017.

[38] Mehdi Kargar, Aijun An, Nick Cercone, Parke Godfrey, Jaroslaw Szlichta, and Xiaohui Yu. Meanks: meaningful keyword search in relational databases with complex schema. In *SIGMOD*, pages 905–908, 2014.

[39] Oliver Lehmberg and Christian Bizer. Stitching web tables for improving matching quality. *PVLDB*, 10(11):1502–1513, 2017.

[40] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. A large public corpus of web tables containing time and context metadata. In *WWW*, pages 75–76, 2016.

[41] Oliver Lehmberg, Dominique Ritze, Petar Ristoski, Robert Meusel, Heiko Paulheim, and Christian Bizer. The mannheim search join engine. *Journal of Web Semantics*, 35:159–166, 2015.

[42] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[43] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. PASS-JOIN: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[44] Girija Limaye, Sunita Sarawagi, and Soumen Chakrabarti. Annotating and searching web tables using entities, types and relationships. *Proc. VLDB Endow.*, 3(1-2):1338–1347, September 2010.

[45] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[46] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.

[47] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[48] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. Unary and n-ary inclusion dependency discovery in relational databases. *J. Intell. Inf. Syst.*, 32(1):53–73, 2009.

[49] Ahmed Metwally and Christos Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

[50] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. Data lake management: Challenges and opportunities (too appear). *PVLDB*, 12, 2019.

[51] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. Table union search on open data. *PVLDB*, 11(7):813–825, 2018.

[52] Thorsten Papenbrock, Sebastian Kruse, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Divide & conquer-based inclusion dependency discovery. *PVLDB*, 8(7):774–785, 2015.

[53] Rakesh Pimplikar and Sunita Sarawagi. Answering table queries on the web using column keywords. In *VLDB*, volume 5, pages 908–919, 2012.

[54] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[55] J Rissanen. Modeling by shortest data description. *Automatica*, 1978.

[56] Chuitian Rong, Chunbin Lin, Yasin N. Silva, Jianguo Wang, Wei Lu, and Xiaoyong Du. Fast and scalable distributed set similarity joins for big data analytics. In *ICDE*, pages 1059–1070, 2017.

[57] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *PVLDB*, 7(12):1059–1070, 2014.

[58] Anish Das Sarma, Lujun Fang, Nitin Gupta, Alon Y. Halevy, Hongrae Lee, Fei Wu, Reynold Xin, and Cong Yu. Finding related tables. In *SIGMOD*, pages 817–828, 2012.

[59] Anshumali Shrivastava and Ping Li. In defense of minhash over simhash. In *AISTATS*, pages 886–894, 2014.

[60] Anshumali Shrivastava and Ping Li. Asymmetric minwise hashing for indexing binary inner products and set containment. In *WWW*, pages 981–991, 2015.

[61] Yasin N. Silva and Jason M. Reed. Exploiting mapreduce-based similarity joins. In *SIGMOD*, pages 693–696, 2012.

[62] Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. Detecting inclusion dependencies on very many tables. *ACM Trans. Database Syst.*, 42(3):18:1–18:29, 2017.

[63] Petros Venetis, Alon Halevy, Jayant Madhavan, Marius Paşca, Warren Shen, Fei Wu, Gengxin Miao, and Chung Wu. Recovering semantics of tables on the web. *Proc. of VLDB Endowment (PVLDB)*, pages 528–538, 2011.

[64] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[65] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.

[66] Jin Wang, Guoliang Li, Dong Deng, Yong Zhang, and Jianhua Feng. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In *ICDE*, pages 519–530, 2015.

[67] Xubo Wang, Lu Qin, Xuemin Lin, Ying Zhang, and Lijun Chang. Leveraging set relations in exact set similarity join. *PVLDB*, 10(9):925–936, 2017.

[68] Robert H. Warren and Frank Wm. Tompa. Multi-column substring matching for database schema translation. *PVLDB*, pages 331–342, 2006.

[69] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[70] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[71] Mohamed Yakout, Kris Ganjam, Kaushik Chakrabarti, and Surajit Chaudhuri. Infogather: Entity augmentation and attribute discovery by holistic matching with web tables. In *SIGMOD*, pages 97–108, 2012.

[72] Erkang Zhu, Yeye He, and Surajit Chaudhuri. Auto-join: Joining tables by leveraging transformations. *PVLDB*, 10(10):1034–1045, 2017.

[73] Erkang Zhu, Dong Deng Fatemeh Nargesian, and Renée J. Miller. JOSIE: Overlap set similarity search for finding joinable tables in data lakes (too appear). In *SIGMOD*, 2019.

[74] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. LSH ensemble: Internet-scale domain search. *PVLDB*, 9(12):1185–1196, 2016.

[75] Erkang Zhu, Ken Q. Pu, Fatemeh Nargesian, and Renée J. Miller. Interactive navigation of open data linkages. *PVLDB*, 10(12):1837–1840, 2017.