Efficient Top-k Algorithms for Fuzzy Search in String Collections

Rares Vernica Chen Li

Department of Computer Science
University of California, Irvine

First International Workshop on Keyword Search on Structured Data, 2009

Outline

1. Motivation
2. Efficient Top-k Algorithms
   - Problem Formulation
   - Algorithms Overview
   - Top-k Single-Pass Search Algorithm
3. Experimental Evaluation

Need for Approximate String Queries

ID	FirstName	LastName	# Movies
10	Al	Swartzberg	1
11	Hanna	Wartenegg	1
12	Rik	Swartzwelder	30
13	Joey	Swartzentruber	1
14	Rene	Swartenbroekx	4
15	Arnold	Schwarzenegger	283
16	Luc	Swartenbroeckx	1
	⋮	⋮	⋮

Figure: Actor names and popularities

Need for Approximate String Queries

ID	FirstName	LastName	# Movies
10	Al	Swartzberg	1
11	Hanna	Wartenegg	1
12	Rik	Swartzwelder	30
13	Joey	Swartzentruber	1
14	Rene	Swartenbroekx	4
15	Arnold	Schwarzenegger	283
16	Luc	Swartenbroeckx	1
	⋮	⋮	⋮

Figure: Actor names and popularities

```
SELECT * FROM Actors
WHERE LastName = 'Shwartzenetrugger'
ORDER BY '# Movies' DESC LIMIT 1;
```

Need for Approximate String Queries

ID	FirstName	LastName	# Movies
10	Al	Swartzberg	1
11	Hanna	Wartenegg	1
12	Rik	Swartzwelder	30
13	Joey	Swartzentruber	1
14	Rene	Swartenbroekx	4
15	Arnold	Schwarzenegger	283
16	Luc	Swartenbroeckx	1
	⋮	⋮	⋮

Figure: Actor names and popularities

```
SELECT * FROM Actors
WHERE LastName = 'Shwartzenetrugger'
ORDER BY '# Movies' DESC LIMIT 1;
```

0 Results

Need for Ranking

ID	FirstName	LastName	# Movies	ED
10	Al	Swartzberg	1	8
11	Hanna	Wartenegg	1	8
12	Rik	Swartzwelder	30	8
13	Joey	Swartzentruber	1	4
14	Rene	Swartenbroekx	4	9
15	Arnold	Schwarzenegger	283	5
16	Luc	Swartenbroeckx	1	9
	⋮	⋮	⋮	⋮

Figure: Actor names, popularities, and edit distances to a query string "Shwartzenetrugger".

Need for Ranking

ID	FirstName	LastName	# Movies	ED
10	Al	Swartzberg	1	8
11	Hanna	Wartenegg	1	8
12	Rik	Swartzwelder	30	8
13	Joey	**Swartzentruber**	1	**4**
14	Rene	Swartenbroekx	4	9
15	Arnold	Schwarzenegger	283	5
16	Luc	Swartenbroeckx	1	9
	⋮	⋮	⋮	⋮

Figure: Actor names, popularities, and edit distances to a query string "Shwartzenetrugger".

Need for Ranking

ID	FirstName	LastName	# Movies	ED
10	Al	Swartzberg	1	8
11	Hanna	Wartenegg	1	8
12	Rik	Swartzwelder	30	8
13	Joey	**Swartzentruber**	1	**4**
14	Rene	Swartenbroekx	4	9
15	Arnold	**Schwarzenegger**	**283**	5
16	Luc	Swartenbroeckx	1	9
	⋮	⋮	⋮	⋮

Figure: Actor names, popularities, and edit distances to a query string "Shwartzenetrugger".

Need for Ranking

ID	FirstName	LastName	# Movies	ED
10	Al	Swartzberg	1	8
11	Hanna	Wartenegg	1	8
12	Rik	Swartzwelder	30	8
13	Joey	**Swartzentruber**	1	**4**
14	Rene	Swartenbroekx	4	9
15	Arnold	**Schwarzenegger**	**283**	5
16	Luc	Swartenbroeckx	1	9
	⋮	⋮	⋮	⋮

Figure: Actor names, popularities, and edit distances to a query string "Shwartzenetrugger".

- **Which one result should the system return?**

Need for Ranking

ID	FirstName	LastName	# Movies	ED
10	Al	Swartzberg	1	8
11	Hanna	Wartenegg	1	8
12	Rik	Swartzwelder	30	8
13	Joey	**Swartzentruber**	1	**4**
14	Rene	Swartenbroekx	4	9
15	Arnold	**Schwarzenegger**	**283**	5
16	Luc	Swartenbroeckx	1	9
	⋮	⋮	⋮	⋮

Figure: Actor names, popularities, and edit distances to a query string "Shwartzenetrugger".

- Which one result should the system return?
- **Which value is more important, # Movies or similarity?**

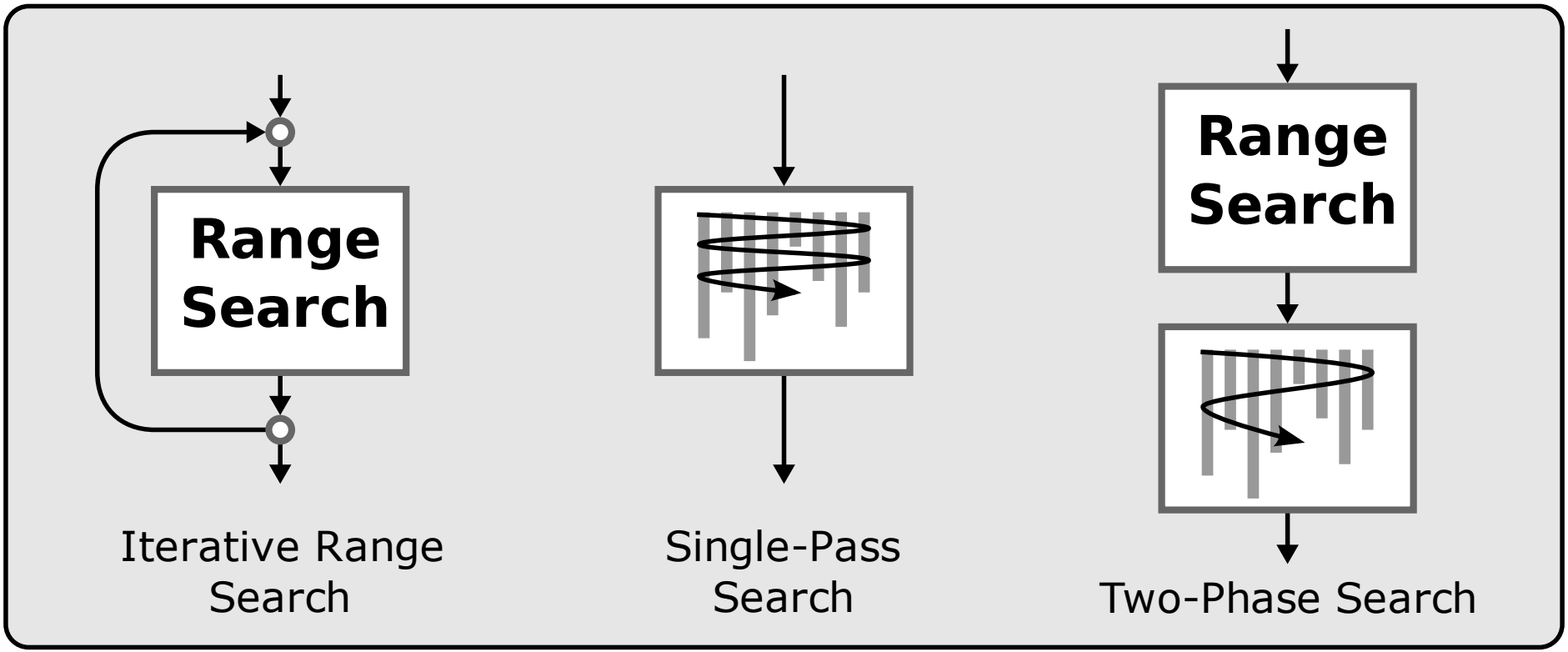
Top-*k* Similar Strings

- Given:
  - Weighted string collection
    - e.g., actors' `LastName` and `# Movies`
  - Query string
    - e.g., "`Shwartzenetrugger`"
  - Similarity function
    - e.g, Edit Distance
  - Scoring function (score of a string in terms of similarity and weight)
    - e.g., linear combination of similarity and popularity
  - Integer *k*
- Return: *k* best strings in terms of overall score to the query string.

Top-*k* Similar Strings

- Given:
  - Weighted string collection
    - e.g., actors' `LastName` and `# Movies`
  - Query string
    - e.g., "`Shwartzenetrugger`"
  - Similarity function
    - e.g, Edit Distance
  - Scoring function (score of a string in terms of similarity and weight)
    - e.g., linear combination of similarity and popularity
  - Integer *k*
- Return: *k* best strings in terms of overall score to the query string.
- **Advantages over Range Search:**
  - specify *k* instead of a similarity threshold
  - guaranteed *k* results; a range search might have **0** results

Algorithms Overview



q-grams: Overlapping substrings of fixed length

- Find similar strings: e.g., "`Vernica`" and "`Veronica`"
- q-gram: substring of length q of a string: e.g., $q = 2$

q-grams: Overlapping substrings of fixed length

- Find similar strings: e.g., "`Vernica`" and "`Veronica`"
- q-gram: substring of length q of a string: e.g., $q = 2$

`Vernica`

q-grams: Overlapping substrings of fixed length

- Find similar strings: e.g., "`Vernica`" and "`Veronica`"
- q-gram: substring of length q of a string: e.g., $q = 2$

`Vernica` $\longrightarrow$ {`Ve`

q-grams: Overlapping substrings of fixed length

- Find similar strings: e.g., "`Vernica`" and "`Veronica`"
- q-gram: substring of length q of a string: e.g., $q = 2$

`Vernica` $\longrightarrow$ {`Ve`,`er`

q-grams: Overlapping substrings of fixed length

- Find similar strings: e.g., "`Vernica`" and "`Veronica`"
- q-gram: substring of length q of a string: e.g., $q = 2$

`Vernica` $\longrightarrow$ {`Ve`,`er`,`rn`

q-grams: Overlapping substrings of fixed length

- Find similar strings: e.g., "`Vernica`" and "`Veronica`"
- q-gram: substring of length q of a string: e.g., $q = 2$

`Vernica` $\longrightarrow$ {`Ve`,`er`,`rn`,`ni`

q-grams: Overlapping substrings of fixed length

- Find similar strings: e.g., "`Vernica`" and "`Veronica`"
- q-gram: substring of length q of a string: e.g., $q = 2$

`Vernica` $\longrightarrow$ {`Ve`,`er`,`rn`,`ni`,`ic`

q-grams: Overlapping substrings of fixed length

- Find similar strings: e.g., "`Vernica`" and "`Veronica`"
- q-gram: substring of length q of a string: e.g., $q = 2$

`Vernica` $\longrightarrow$ `{Ve,er,rn,ni,ic,ca}`

q-grams: Overlapping substrings of fixed length

- Find similar strings: e.g., "`Vernica`" and "`Veronica`"
- q-gram: substring of length q of a string: e.g., $q = 2$

`Vernica` $\longrightarrow$ {`Ve`, `er`, `rn`, `ni`, `ic`, `ca`}

`Veronica` $\longrightarrow$ {`Ve`, `er`, `ro`, `on`, `ni`, `ic`, `ca`}

q-grams: Overlapping substrings of fixed length

- Find similar strings: e.g., "Vernica" and "Veronica"
- q-gram: substring of length q of a string: e.g., $q = 2$

Vernica $\longrightarrow$ {**Ve**, **er**, rn, **ni**, **ic**, **ca**}

Veronica $\longrightarrow$ {**Ve**, **er**, ro, on, **ni**, **ic**, **ca**}

- **Similar strings share a certain number of grams**

q-gram Inverted List Index

- $q = 2$

ID	String	Weight
1	ab	0.80
2	ccd	0.70
3	cd	0.60
4	abcd	0.50
5	bcc	0.40

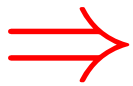
Figure: Dataset

q-gram Inverted List Index

- $q = 2$

ID	String	Weight
1	ab	0.80
2	ccd	0.70
3	cd	0.60
4	abcd	0.50
5	bcc	0.40

Figure: Dataset



ab	cc	cd	bc
1	2	2	4
4	5	3	5
		4	

Figure: Gram inverted-list index

q-gram Inverted List Index

- $q = 2$

ID	String	Weight
1	ab	0.80
2	ccd	0.70
3	cd	0.60
4	abcd	0.50
5	bcc	0.40

Figure: Dataset

ab	cc	cd	bc
1	2	2	4
4	5	3	5
		4	

Figure: Gram inverted-list index

- Query string: **"bcd"**

q-gram Inverted List Index

- $q = 2$

ID	String	Weight
1	ab	0.80
2	ccd	0.70
3	cd	0.60
4	abcd	0.50
5	bcc	0.40

Figure: Dataset

ab	cc	cd	bc
1	2	2	4
4	5	3	5
		4	

Figure: Gram inverted-list index

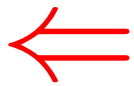
- Query string: "bcd"

q-gram Inverted List Index

- $q = 2$

ID	String	Weight
1	ab	0.80
2	**ccd**	0.70
3	**cd**	0.60
4	**abcd**	0.50
5	**bcc**	0.40

Figure: Dataset



ab	cc	cd	bc
1	2	**2**	**4**
4	5	**3**	**5**
		4	

Figure: Gram inverted-list index

- Query string: **"bcd"**
- Identified strings are verified by computing the real similarity. Verification is usually an **expensive** process.

Top-k Single-pass Search Algorithm

ID	String	Weight
1	ab	0.80
2	ccd	0.70
3	cd	0.60
4	abcd	0.50
5	bcc	0.40

Figure: Dataset

ab	cc	cd	bc
1	2	2	4
4	5	3	5
		4	

Figure: Gram inverted-list index

Top-k Single-pass Search Algorithm

Setup

- Assign IDs s.t.
  ascending order of IDs $\equiv$
  descending order of weights

ID	String	Weight
1	ab	0.80
2	ccd	0.70
3	cd	0.60
4	abcd	0.50
5	bcc	0.40

Figure: Dataset

ab	cc	cd	bc
1	2	2	4
4	5	3	5
		4	

Figure: Gram inverted-list index

Top-k Single-pass Search Algorithm

Setup

- Assign IDs s.t.
  ascending order of IDs $\equiv$
  descending order of weights
- Sort the IDs on each list in ascending order

ID	String	Weight
1	ab	0.80
2	ccd	0.70
3	cd	0.60
4	abcd	0.50
5	bcc	0.40

Figure: Dataset

ab	cc	cd	bc
1	2	2	4
4	5	3	5
		4	

Figure: Gram inverted-list index

Top-k Single-pass Search Algorithm

Setup

- Assign IDs s.t.
  ascending order of IDs $\equiv$
  descending order of weights
- Sort the IDs on each list in ascending order
- Scan the lists corresponding to the grams in the query.
  e.g., **"bcd"**

ID	String	Weight
1	ab	0.80
2	ccd	0.70
3	cd	0.60
4	abcd	0.50
5	bcc	0.40

Figure: Dataset

ab	cc	cd	bc
1	2	2	4
4	5	3	5
		4	

Figure: Gram inverted-list index

Top-k Single-pass Search Algorithm

Naïve approach: Round-Robin

- Scan all the lists in the same time
- Maintain a list of "open" IDs (might still appear on some of the lists)
- Store the best k "closed" IDs in a top-k buffer
- **Stop** when the top-k buffer cannot improve

ID	String	Weight
1	ab	0.80
2	ccd	0.70
3	cd	0.60
4	abcd	0.50
5	bcc	0.40

Figure: Dataset

ab	cc	cd	bc
1	2	2	4
4	5	3	5
		4	

Figure: Gram inverted-list index

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
1	2	2
20	3	4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
→1	2	2
20	3	4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

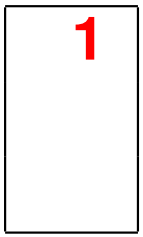


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
→1	→**2**	2
20	3	4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

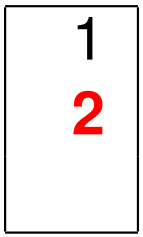


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
→1	→2	→**2**
20	3	4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

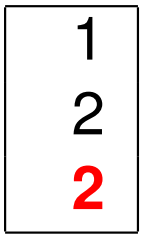


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
→1	→2	→2
20	3	4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

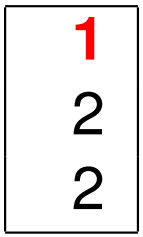


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. **No need to maintain the list of "open" IDs**
  2. Skip elements

ab	bc	cd
→1	→2	→2
20	3	4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

Figure: Top-k buffer, $k = 1$

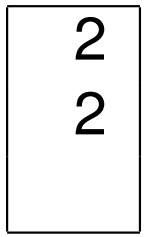


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
1	→2	→2
→**20**	3	4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

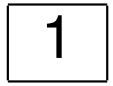


Figure: Top-k buffer, $k = 1$

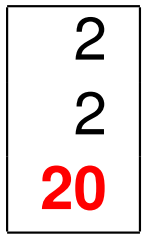


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
1	→2	→2
→20	3	4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

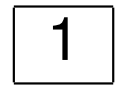


Figure: Top-k buffer, $k = 1$

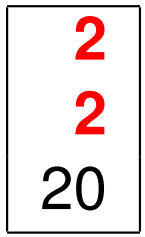


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
1	→2	→2
→20	3	4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

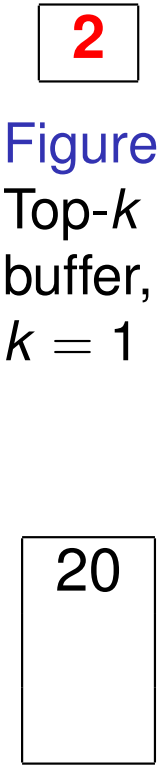


Figure: Top-k buffer, $k = 1$

Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
1	2	2
→20	→**3**	→**4**
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

Figure: Top-k buffer, $k = 1$

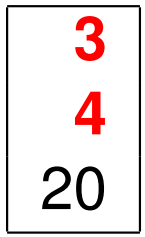


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
1	2	2
→20	→3	→4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

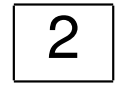


Figure: Top-k buffer, $k = 1$

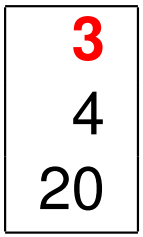


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
1	2	2
→20	→3	→4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

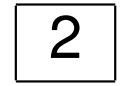


Figure: Top-k buffer, $k = 1$

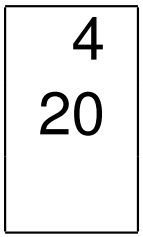


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
1	2	2
→20	→3	→4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

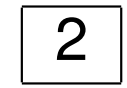


Figure: Top-k buffer, $k = 1$

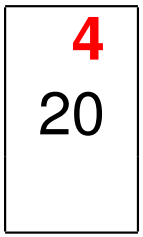


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. Skip elements

ab	bc	cd
1	2	2
→20	→3	→4
21	4	5
⋮	⋮	⋮
	19	19
	20	20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

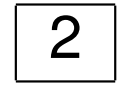


Figure: Top-k buffer, $k = 1$

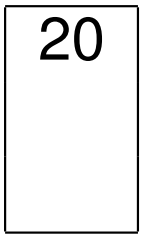


Figure: Min-heap

Top-k Single-pass Search Algorithm

Heap-based

- Traverse the lists in a sorted order using a heap on the top IDs of the lists
- Advantages:
  1. No need to maintain the list of "open" IDs
  2. **Skip elements**

ab	bc	cd
1	2	2
→20	3	4
21	4	5
⋮	⋮	⋮
	19	19
	→20	→20
	⋮	⋮

Figure: Gram inverted-lists for query "abcd"

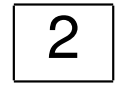


Figure: Top-k buffer, $k = 1$

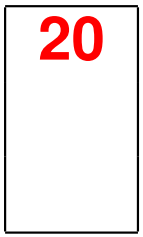
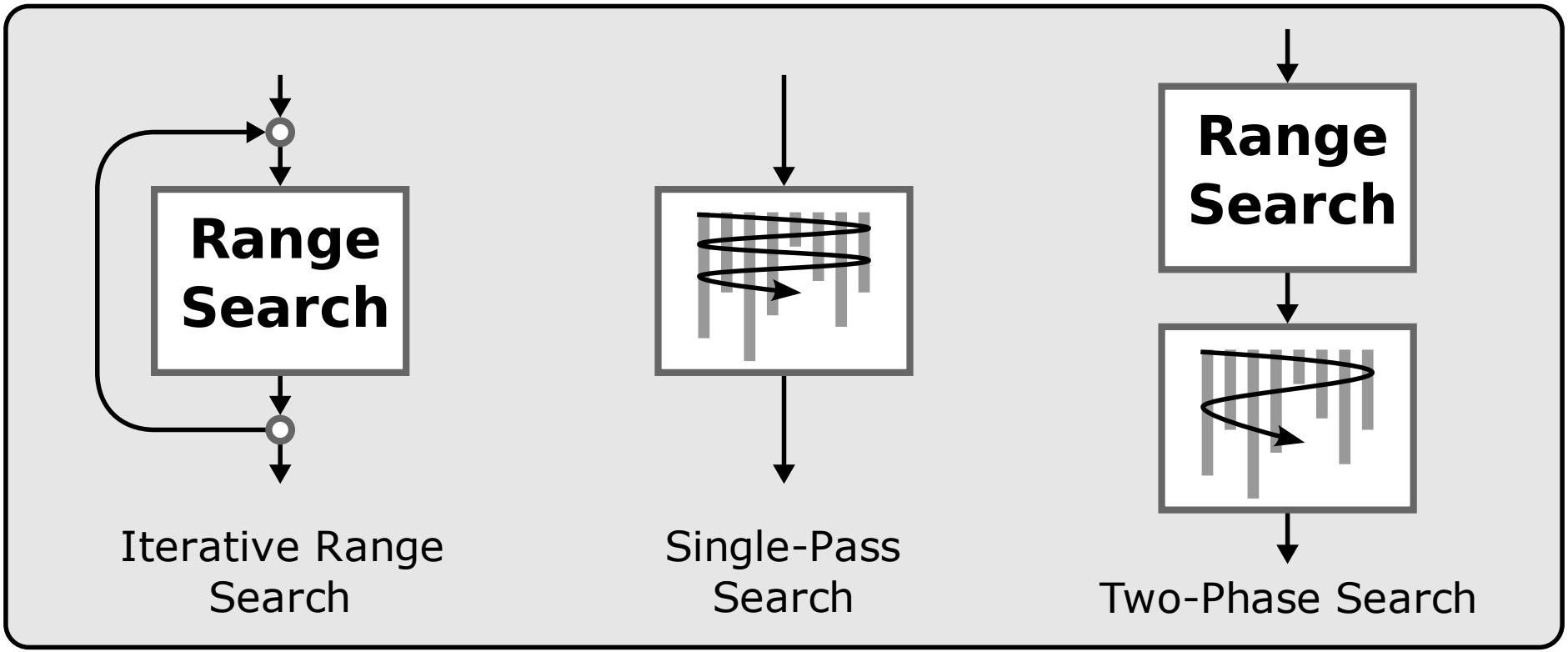


Figure: Min-heap

Algorithms Overview



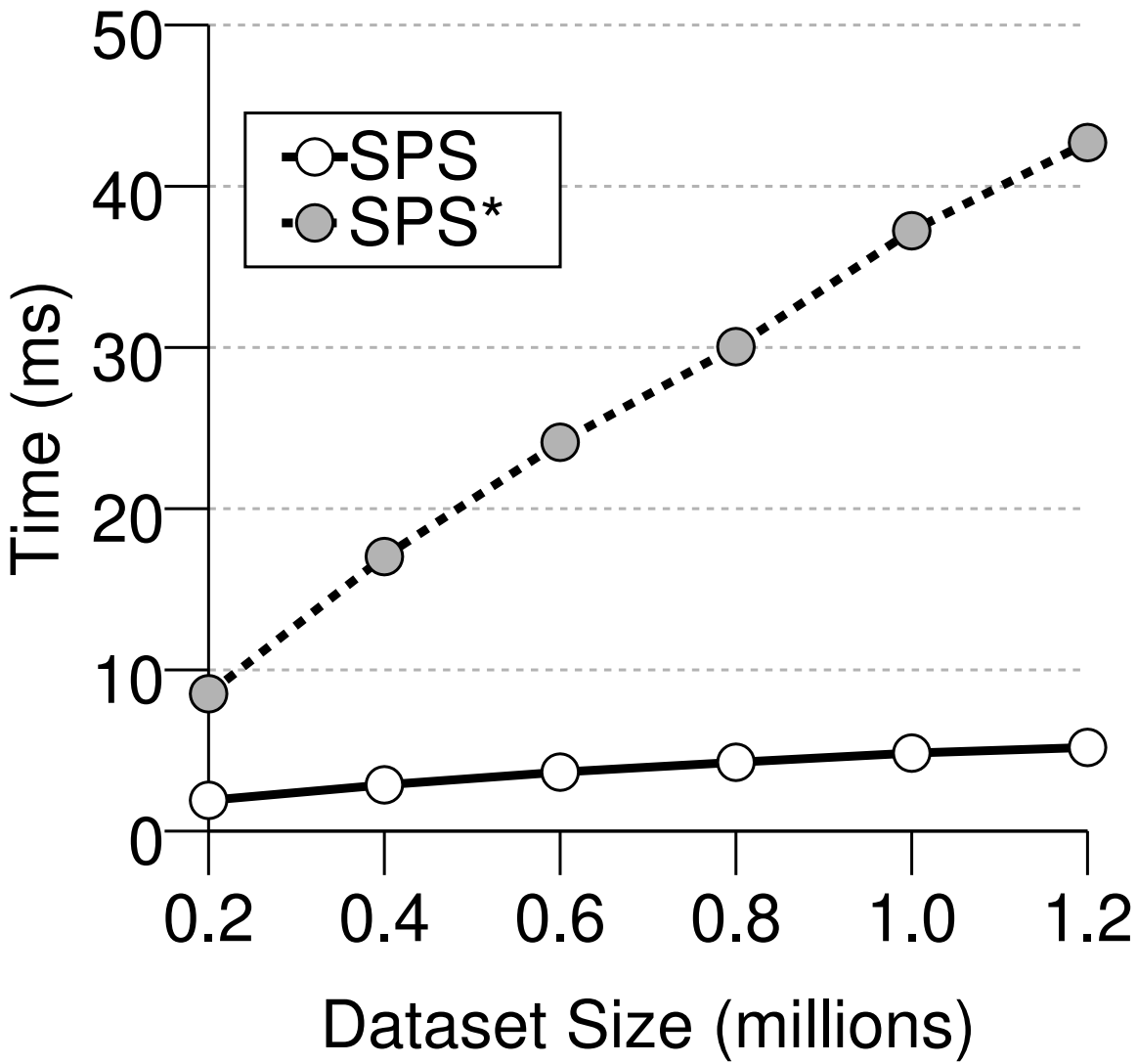
Experimental Setting

- Datasets:
  - IMDB Actor Names[1]
    - actor names and the number of movies they played in
    - 1.2 million actors, average name length 15
    - weight is the number of movies (log normalized)
  - WEB Corpus Word Grams[2]
    - sequences of English words and their frequency on the Web
    - 2.4 million sequences, average sequence length 20
    - weight is the frequency (log normalized)
- Jaccard similarity and normalized edit similarity, $q = 3$
- Index and data are stored in main memory at all times
- Implemented in C++ (GNU compiler) on Ubuntu Linux OS
- Intel 2.40GHz PC, 2GB RAM

[1] `http://www.imdb.com/interfaces`

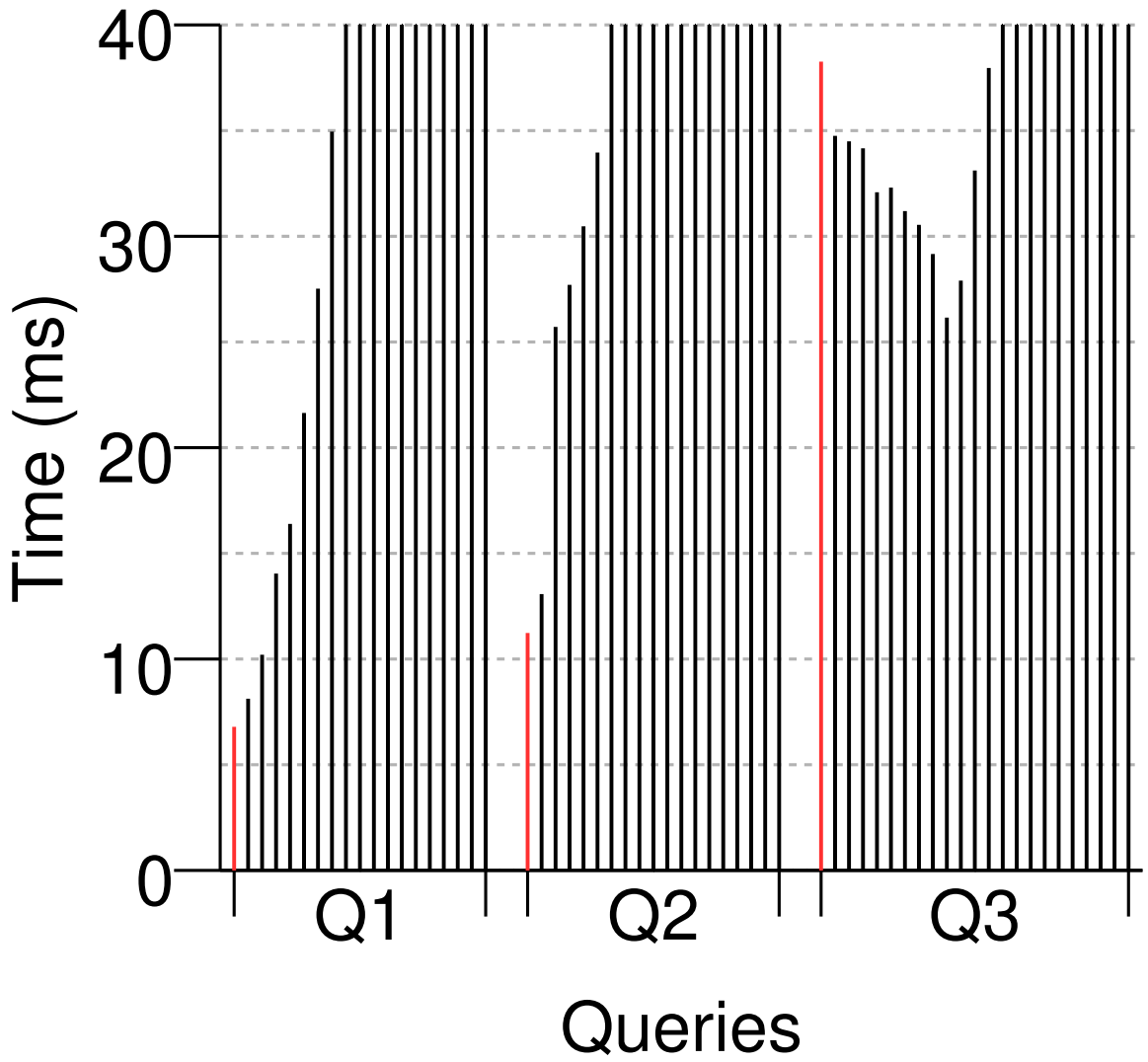
[2] `http://www.ldc.upenn.edu/Catalog` LDC2006T13

Benefits of Skipping Elements



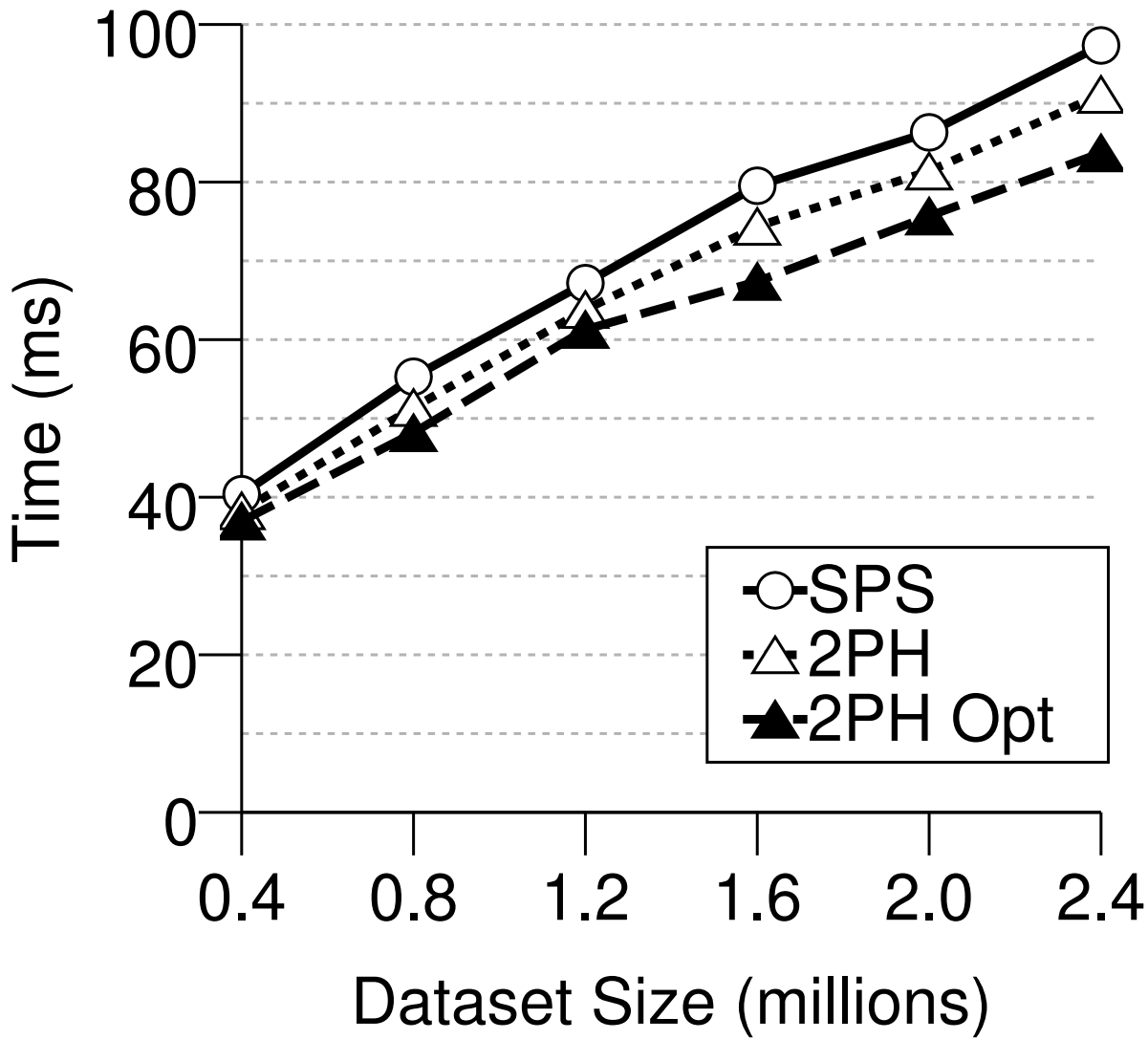
Average running time for top-10 queries. IMDB dataset with Jaccard similarity. Single-Pass Search (SPS) algorithm and SPS without skipping (SPS*).

Potential of the Two-Phase Algorithm



Running time for 3 top-10 queries. WEB Corpus dataset with normalized edit similarity. Two-Phase algorithm with different initial thresholds.

Optimum Initial Threshold for the Two-Phase Algorithm



Average running time for top-10 queries. Web Corpus dataset with normalized edit similarity. Single-Pass Search (SPS) algorithm, Two-Phase (2PH) algorithm, 2PH with the optimum initial threshold (2PH Opt).

The Iterative Range Search algorithm was to expensive to be plotted. The average running time was around 5s.

Summary

- Approximate ranking queries in string collections
- Useful when mismatch between query and data
- Propose three approaches to solve the problem:
  1. Use existing approximate range search algorithms as a “black box”
     Proves to be the most expensive
  2. Use particularities of the top-k problem
     Proves to be very efficient
  3. Combine (1) and (2) sequentially
     Proves to be slightly more efficient

The Flamingo Project

This work is part of
The Flamingo Project at UC Irvine
`http://flamingo.ics.uci.edu`

A Quick Note on Related Work

Fagin et. al [1]

- similarity on multiple numerical attributes
- traverse list of IDs
- one list per attribute
- lists sorted on similarity to that attribute
- lists have different orders of IDs
- all IDs appear on all the lists

Our Setting

- similarity on one string attribute
- traverse list of IDs
- one list per q-gram
- lists sorted on global **weight**
- lists have the **same** order of IDs
- a **subset** of IDs appear on each list

[1] R.Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In PODS, 2001