Similarity Search in Large Databases

Introduction

Nikolaus Augsten
nikolaus.augsten@sbg.ac.at
Department of Computer Sciences
University of Salzburg

database research group
http://dbresearch.uni-salzburg.at

WS 2019/20

Version November 13, 2019

A Problem at Our Municipality of Bozen

- Given:
  - reality owners DB (name and address of the reality)
  - residents DB (name and residential address)
  - both DBs cover the same geographic area (the city of Bozen)

Owners (dataset A)

Peter	Gilmstrasse 1
Arturas	Gilmstrasse 3
Linas	Marieng. 1/A
Markus	Cimitero 4
Michael	Gilmstrasse 5
Igor	Friedensplatz 2/A/1
Andrej	Friedensplatz 3
Francesco	Untervigil 1
Johann	Cimitero 6/B
Igor	Friedensplatz 2/A/2
Nikolaus	Cimitero 6/A

Residents (dataset B)

Rosa	Siegesplatz 3/-/3
Dario	Friedhofplatz 4
Romans	Untervigli 1
Adriano	Mariengasse 1
Maria	Siegesplatz 3/-/2
Arturas	Hermann-von-Gilm-Str. 3/A
Peter	Hermann-von-Gilm-Str. 1
Markus	Siegesplatz 2/A
Juozas	Hermann-von-Gilm-Str. 3/B
Andrej	Siegesplatz 3/-/1
Luigi	Friedhofplatz 6
Anita	Herman-von-Gilm-Str. 6

- Query: Give me owner and resident for each apartment in Bozen!

Outline

Outline

All Information about Lecture and Lab

https://dbresearch.uni-salzburg.at/teaching/2019ws/ssdb/

Outline

What is Similarity Search?

- Similarity search deals with the question:

$$\boxed{\text{How similar are two objects?}}$$

- "Objects" may be
  - strings (`Augsten` $\leftrightarrow$ `Augusten`)
  - tuples in a relational database

$$(\texttt{Augsten} \mid \texttt{Dominikanerplatz 3} \mid \texttt{204} \mid \texttt{70188})$$
$$\leftrightarrow$$
$$(\texttt{N. Augsten} \mid \texttt{Dominikanerpl. 3} \mid \texttt{@} \mid \texttt{70188})$$

  - documents (e.g., HTML or XML)
  - ...
- "Similar" is application dependant

Application I: Object Identification

- Problem:
  - Two data items represent the same real world object (e.g., the same person),
  - but they are represented differently in the database(s).
- How can this happen?
  - different coding conventions (e.g., `Gilmstrasse`, `Hermann-von-Gilm-Str.`)
  - spelling mistakes (e.g., `Untervigil`, `Untervigli`)
  - outdated values (e.g., `Siegesplatz` used to be `Friedensplatz`).
  - incomplete/incorrect values (e.g., missing or wrong apartment number in residential address).
- Focus in this course!

Application I: Flavors of Object Identification

- Duplicate Detection
  - one table
  - find all tuples in the table that represent the same thing in the real world
  - Example: Two companies merge and must build a single customer database.
- Similarity Join
  - two tables
  - join all tuples with similar values in the join attributes
  - Example: In order to detect tax fraud, data from different databases need to be linked.
- Similarity Lookup
  - one table, one tuple
  - find the tuple in the table that matches the given tuple best
  - Example: Do we already have customer X in the database?

Application II: Computational Biology

- DNA and protein sequences
  - modelled as text over alphabet (e.g. $\{A, C, G, T\}$ in DNA)
- Application: Search for a pattern in the text
  - look for given feature in DNA
  - compare two DNAs
  - decode DNA
- Problem: Exact matches fail
  - experimental measures have errors
  - small changes that are not relevant
  - mutations
- Solution: Similarity search
  - Search for *similar* patterns
  - *How similar* are the patterns that you found?

Application III: Error Correction in Signal Processing

- Application: Transmit text signal over physical channel
- Problem: Transmission may introduce errors
- Goal: Restore original (sent) message
- Solution: Find correct text that is closest to received message.

Framework for Similarity Search

1. Preprocessing (e.g., lowercase `Augsten` → `augsten`)
2. Search Space Reduction
    - Blocking
    - Sorted-Neighborhood
    - Filtering (Pruning)
3. Compute Distances
4. Find Matches

Search Space Reduction: Brute Force

- We consider the example of similarity join.
- Similarity Join: Find all pairs of similar tuples in tables A and B.
  - Search space: $A \times B$ (all possible pairs of tuples)
  - Complexity: compute $|A||B|$ distances $\rightarrow$ expensive!
    ($|A| = 30k$, $|B| = 40k$, $1ms$ per distance $\Rightarrow$ join runs 2 weeks)
- Example: 16 distance computations!

A

Tim	m
Bill	m
Jane	f
Mary	f

B

Bil	m
Jane	f
Tim	m
Marie	f

- Goal: Reduce search space!

Search Space Reduction: Blocking

- Blocking
  - Partition A and B into blocks (e.g., group by chosen attribute).
  - Compare only tuples within blocks.
- Example: Block by gender (m/f):

Tim	m
Bill	m

Bil	m
Tim	m

Mary	f
Jane	f

Jane	f
Marie	f

- Improvement: 8 distance computations (instead of 16)!

Search Space Reduction: Sorted Neighborhood

- Sorted Neighborhood
  - Sort A and B (e.g., by one of the attributes).
  - Move a window of fixed size over A and B.
    - move A-window if sort attribute of next tuple in A is smaller than in B
    - otherwise move B-window
  - Compare only tuples within the windows.
- Example: Sort by name, use window of size 2:

A

Bill	m
Jane	f
Mary	f
Tim	m

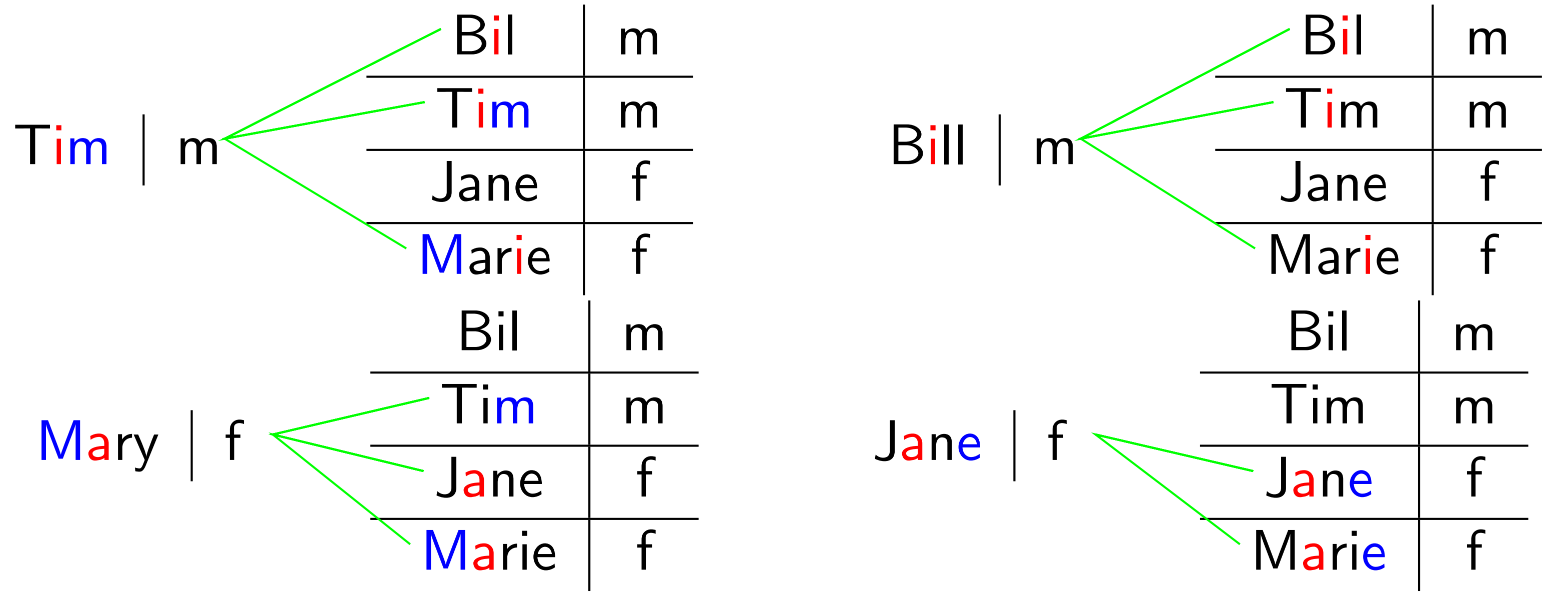
B

Bil	m
Jane	f
Marie	f
Tim	m

- Improvement: 12 distance computations (instead of 16)!

Search Space Reduction: Filtering

- Filtering (Pruning)
  - Remove (filter) tuples that cannot match, then compute the distances.
  - Idea: filter is faster than distance function.
- Example: Do not match names that have no character in common:



- Improvement: 11 distance computations (instead of 16)!

Distance Computation

Definition (Distance Function)

Given two sets of objects, A and B, a *distance function* for A and B maps each pair $(a, b) \in A \times B$ to a positive real number (including zero).

$$\delta : A \times B \to \mathbb{R}_0^+$$

- We will define distance functions for
  - sets
  - strings
  - ordered, labeled trees
  - unordered, labeled trees

Distance Matrix

Definition (Distance Matrix)

Given a distance function δ for two sets of objects, $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$.
The *distance matrix* D is an $n \times m$-matrix with

$$d_{ij} = \delta(a_i, b_j),$$

where d_{ij} is the element at the i-th row and the j-th column of D.

- Example distance matrix, $A = \{a_1, a_2, a_3\}$, $B = \{b_1, b_2, b_3\}$:

	b_1	b_2	b_3
a_1	6	5	4
a_2	2	2	1
a_3	1	3	0

Finding Matches: Threshold

	b_1	b_2	b_3
a_1	6	5	4
a_2	2	2	1
a_3	1	3	0

- Once we know the distances – which objects match?
- Threshold Approach:
  - fix threshold τ
  - algorithm:

    foreach $d_{ij} \in D$ **do**
        if $d_{ij} < \tau$ **then** match (a_i, b_j)
  - produces n:m-matches
- Example with $\tau = 3$: $\{(a_2, b_1), (a_2, b_2), (a_2, b_3), (a_3, b_1), (a_3, b_3)\}$

Finding Matches: Global Greedy

- Global Greedy Approach:
  - algorithm:

    $M \leftarrow \emptyset$
    create sorted list L with all $d_{ij} \in D$
    while $A \neq \emptyset$ **and** $B \neq \emptyset$ **do**
        $d_{ij} \leftarrow$ deque smallest element from L
        if $a_i \in A$ **and** $b_j \in B$ **then**
            $M \leftarrow M \cup (a_i, b_j)$
            remove a_i from A and b_j from B
    return M
  - produces 1:1-matches
  - must deal with tie distances when sorting L! (e.g. sort randomly, sort by i and j)
- Example (sort ties by i, j):
  $\{(a_3, b_3), (a_2, b_1), (a_1, b_2)\}$

	b_1	b_2	b_3
a_1	6	5	4
a_2	2	2	1
a_3	1	3	0

Overview: Finding Matches

	b_1	b_2	b_3
a_1	6	5	4
a_2	2	2	1
a_3	1	3	0

- Threshold Approach:
  - all objects with distance below τ match
  - produces n:m-matches
  - threshold approach for our example with $\tau = 3$:
    $\{(a_2, b_1), (a_2, b_2), (a_2, b_3), (a_3, b_1), (a_3, b_3)\}$
- Global Greedy Approach:
  - pair with smallest distance is chosen first
  - produces 1:1-matches
  - global greedy approach for our example:
    $\{(a_3, b_3), (a_2, b_1), (a_1, b_2)\}$

Assumptions for the Solutions in this Course

- Large data volumes
  - cannot be done by hand
  - solution must be efficient
- Data-driven, not process-driven
  - Sometimes it is better to change the world, e.g., force people to adhere to coding conventions, instead of fixing the errors later.
  - We cannot change the world.
- No domain-specific solution (e.g., address standardization)
- No training phase (e.g., supervised learning)
- No expensive configuration (e.g., define dictionaries, rules)
- Tuning parameters (like weights) are OK

Outline

Back to Our Initial Example

- Given:
  - reality owners DB (name and address of the reality)
  - residents DB (name and residential address)
  - both DBs cover the same geographic area (the city of Bozen/Italy)

Owners (dataset A)

Peter	Gilmstrasse 1
Arturas	Gilmstrasse 3
Linas	Marieng. 1/A
Markus	Cimitero 4
Michael	Gilmstrasse 5
Igor	Friedensplatz 2/A/1
Andrej	Friedensplatz 3
Francesco	Untervigil 1
Johann	Cimitero 6/B
Igor	Friedensplatz 2/A/2
Nikolaus	Cimitero 6/A

Residents (dataset B)

Rosa	Siegesplatz 3/-/3
Dario	Friedhofplatz 4
Romans	Untervigli 1
Adriano	Mariengasse 1
Maria	Siegesplatz 3/-/2
Arturas	Hermann-von-Gilm-Str. 3/A
Peter	Hermann-von-Gilm-Str. 1
Markus	Siegesplatz 2/A
Juozas	Hermann-von-Gilm-Str. 3/B
Andrej	Siegesplatz 3/-/1
Luigi	Friedhofplatz 6
Anita	Herman-von-Gilm-Str. 6

- Give me owner and resident for each apartment in Bozen!

Database Representation

Owners

A

strID	*name*	*num*	*entr*	*apt*
α_1	Gilmstrasse	1		
α_1	Gilmstrasse	3		
α_1	Gilmstrasse	5		
α_2	Fiedensplatz	2	A	1
α_2	Fiedensplatz	2	A	2
α_2	Fiedensplatz	3		
α_3	Cimitero	4		
α_3	Cimitero	6	A	
α_3	Cimitero	6	B	
α_4	Untervigil	1		
α_5	Marieng.	1	A	

Residents

B

strID	*name*	*num*	*entr*	*apt*
β_2	Hermann-von-Gilm-Str.	1		
β_2	Hermann-von-Gilm-Str.	3	A	
β_2	Hermann-von-Gilm-Str.	3	B	
β_2	Hermann-von-Gilm-Str.	6		
β_3	Siegesplatz	2	A	
β_3	Siegesplatz	3	-	1
β_3	Siegesplatz	3	-	2
β_3	Siegesplatz	3	-	3
β_1	Friedhofplatz	4		
β_1	Friedhofplatz	6		
β_5	Untervigli	1		
β_4	Mariengasse	1		

String Similarity

- **Observation 1:** Some street names are similar.

dataset A	dataset B
Gilmstrasse	Friedhofplatz
Friedensplatz	Hermann-von-Gilm-Str.
Cimitero	Siegesplatz
Untervigil	Mariengasse
Marieng.	Untervigli

- We match:
  - Untervigil ↔ Untervigli
  - Marieng. ↔ Mariengasse
  - Gilmstrasse ↔ Hermann-von-Gilm-Str.
- But what to do with the others?
  - Friedensplatz was renamed to Siegesplatz, but one database was not updated
  - Cimitero is the Italian name for Friedhofplatz (German name)
- Problem: Friedensplatz looks more like Friedhofplatz than like Siegesplatz!

Demo: String Similarity

- Street name tables:

strID	*name*
α_1	`Gilmstrasse`
α_2	`Friedensplatz`
α_3	`Cimitero`
α_4	`Untervigil`
α_5	`Marieng.`

strID	*name*
β_1	`Friedhofplatz`
β_2	`Hermann-von-Gilm-Str.`
β_3	`Siegesplatz`
β_4	`Mariengasse`
β_5	`Untervigli`

- Distance matrix for the *q*-gram distance between strings:

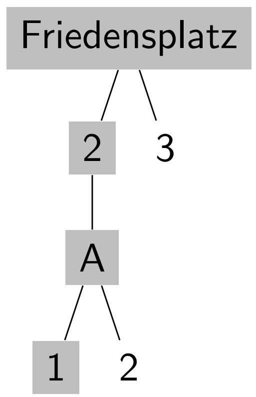
	β_1	β_2	β_3	β_4	β_5
α_1	1.0	0.8333	1.0	0.6923	1.0
α_2	0.3333	1.0	0.5714	0.9286	1.0
α_3	1.0	1.0	1.0	1.0	0.9091
α_4	1.0	0.9429	1.0	1.0	0.3333
α_5	0.92	0.9394	1.0	0.3913	1.0

- Matches with the global greedy algorithm:
  $\{(\alpha_2, \beta_1), (\alpha_4, \beta_5), (\alpha_5, \beta_4), (\alpha_1, \beta_2), (\alpha_3, \beta_3), \}$

Tree Similarity

- **Observation 2**: Different streets have different addresses.
- Build *address tree*:

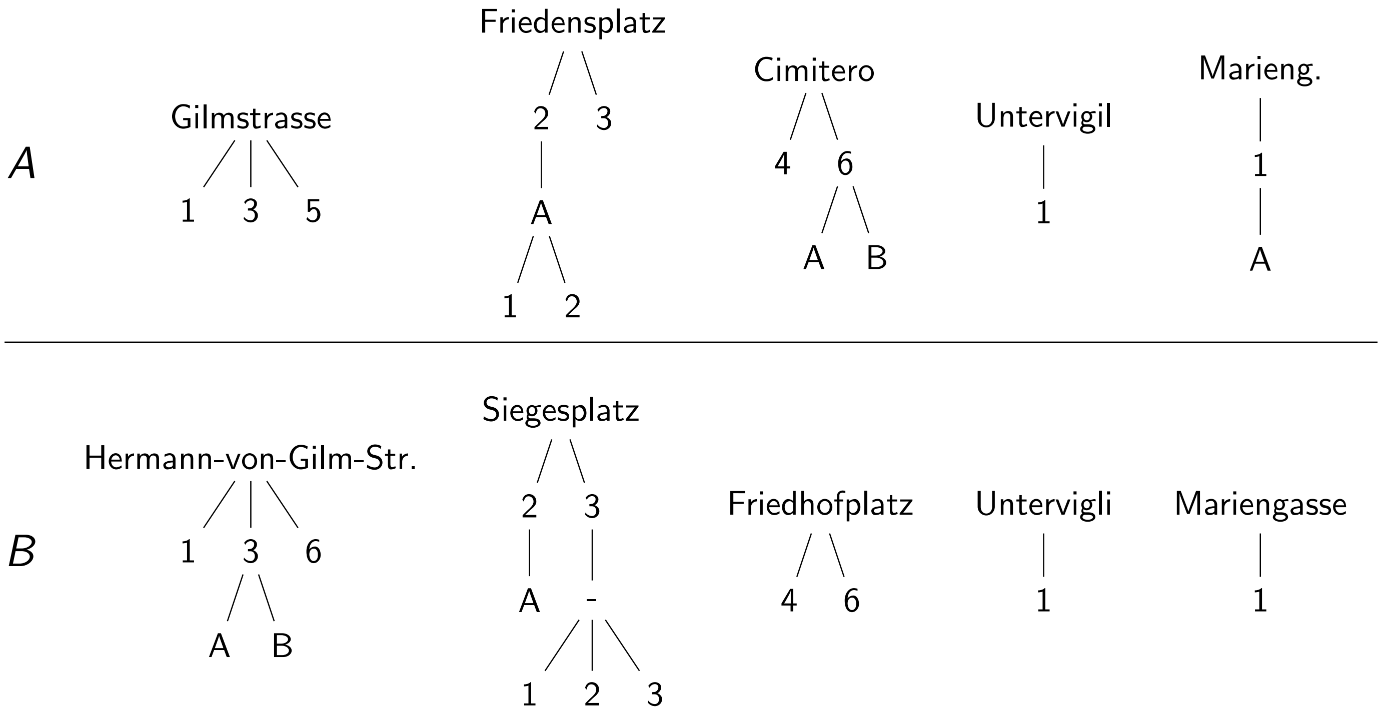
```
Gilmstrasse 1
Gilmstrasse 3
Gilmstrasse 5
Friedensplatz 2/A/1
Friedensplatz 2/A/2
Friedensplatz 3
Cimitero 4
Cimitero 6/A
Cimitero 6/B
Untervigil 1
Marieng. 1/A
```



- Address is path from root to leaf.
- Example: Shaded path is the address `Friedensplatz 2/A/1` (house number `2`, entrance `A`, apartment `1`).

Tree Similarity

- Address trees of our example:



- Ignore root labels for distance computation.
- Trees of `Siegesplatz` and `Friedensplatz` are similar :-)
- Trees of `Cimitero` and `Friedhofplatz` are similar :-)
- But: `Untervigil` and `Mariengasse` have identical address trees in dataset B.

Demo: Tree Similarity

- Street name tables:

strID	*name*
α_1	`Gilmstrasse`
α_2	`Friedensplatz`
α_3	`Cimitero`
α_4	`Untervigil`
α_5	`Marieng.`

strID	*name*
β_1	`Friedhofplatz`
β_2	`Hermann-von-Gilm-Str.`
β_3	`Siegesplatz`
β_4	`Mariengasse`
β_5	`Untervigli`

- Distance matrix for the *pq*-gram distance between trees:

	β_1	β_2	β_3	β_4	β_5
α_1	1.0	0.7143	1.0	0.6667	0.6667
α_2	1.0	1.0	0.5758	1.0	1.0
α_3	0.4118	0.9167	1.0	1.0	1.0
α_4	1.0	0.7647	1.0	0.0	0.0
α_5	1.0	0.9	1.0	0.4545	0.4545

- Matches with the global greedy algorithm:
$\{(\alpha_4, \beta_4), (\alpha_3, \beta_1), (\alpha_5, \beta_5), (\alpha_2, \beta_3), (\alpha_1, \beta_2)\}$

Combining String and Tree Distance

- Use strings *and* trees!
- String distance s, tree distance t
- Weight $\omega \in [0..1]$
  - $\omega = 0 \rightarrow$ only trees
  - $\omega = 1 \rightarrow$ only strings
- overall distance d (using weighted Euclidean distance):

$$d = \sqrt{\omega s^2 + (1 - \omega) t^2}$$

Demo: Combining String and Tree Distance

- Computed with $w = 0.5$ from string and tree matrices:

	β_1	β_2	β_3	β_4	β_5
α_1	1.0	0.7761	1.0	0.6796	0.8498
α_2	0.7454	1.0	0.5736	0.9649	1.0
α_3	0.7647	0.9592	1.0	1.0	0.9556
α_4	1.0	0.8584	1.0	0.7071	0.2357
α_5	0.9608	0.9199	1.0	0.4241	0.7767

- Matches with the global greedy algorithm:
  $\{(\alpha_4, \beta_5), (\alpha_5, \beta_4), (\alpha_2, \beta_3), (\alpha_3, \beta_1), (\alpha_1, \beta_2)\}$

- All matches are correct :-)

`Gilmstrasse` ↔ `Hermann-von-Gilm-Str.`
`Friedensplatz` ↔ `Siegesplatz`
`Cimitero` ↔ `Friedhofplatz`
`Untervigil` ↔ `Untervigli`
`Marieng.` ↔ `Mariengasse`

`map_A_B`

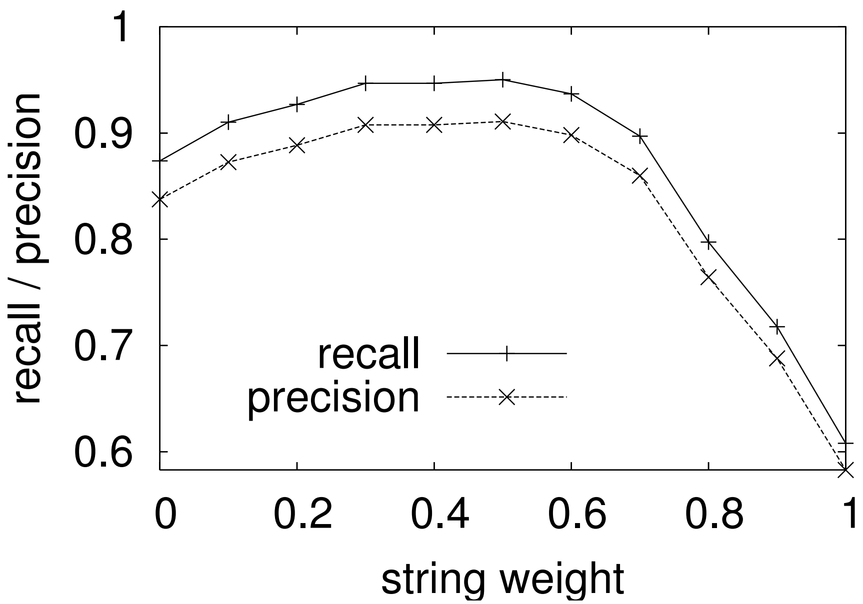
idA	*idB*
α_4	β_5
α_5	β_4
α_2	β_3
α_3	β_1
α_1	β_2

Experiments: Results for Real World Data

- Similarity join on three real databases:
  - electricity company (elec) – German street names, 45k addresses
  - registration office (reg) – Italian street names, 43k addresses
  - census database (cens) – German street names, 11k addresses
- Measure precision and recall
  - Precision: correctly computed matches to total number of computed matches
  - Recall: correctly computed matches to total number of correct matches

Experiments: Results for Real World Data

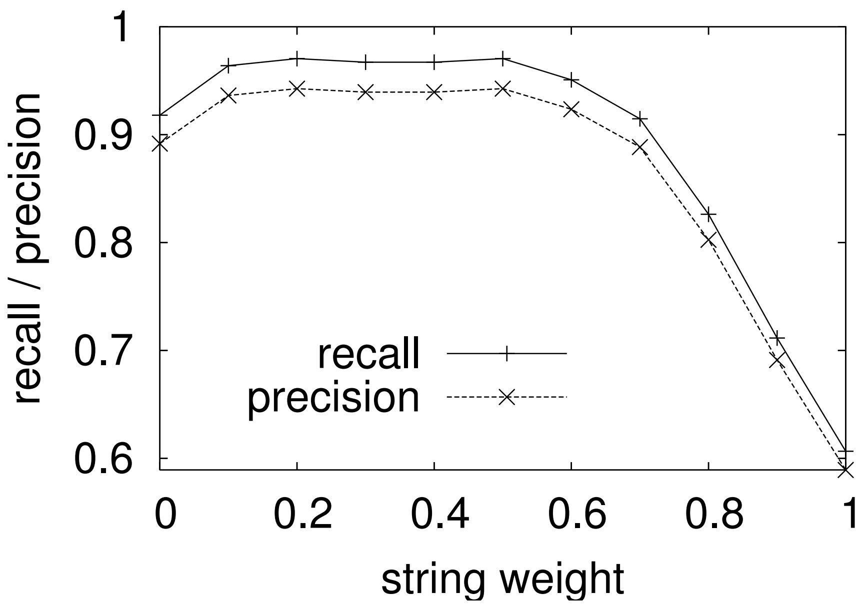
- Similarity join with global greedy matching
- String weight ω varies from 0 (only trees) to 1 (only strings)
- Measure precision and recall (high is good)



elec (German) $\leftrightarrow$ reg (Italian)

Experiments: Results for Real World Data

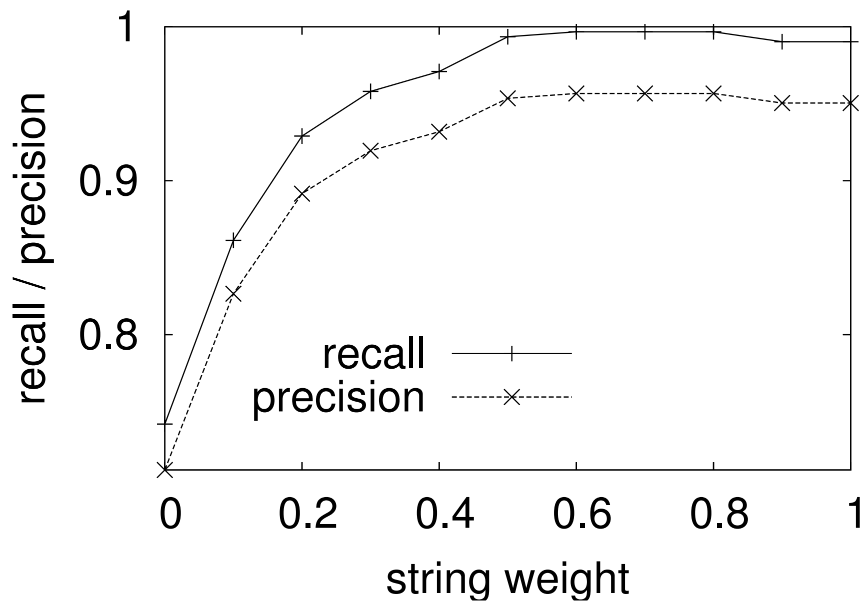
- Similarity join with global greedy matching
- String weight ω varies from 0 (only trees) to 1 (only strings)
- Measure precision and recall (high is good)



reg (Italian) $\leftrightarrow$ cens (German)

Experiments: Results for Real World Data

- Similarity join with global greedy matching
- String weight ω varies from 0 (only trees) to 1 (only strings)
- Measure precision and recall (high is good)



elec (German) $\leftrightarrow$ reg (German)

Experiments: Results for Real World Data

Summary of the experimental results:

- High string weight ω good for German-German, bad for German-Italian
- String weight $\omega = 0.5$ good for both German-German and German-Italian
- Precision and recall very high ($\omega = 0.5$):
  - more than 90% even for German-Italian
  - precision almost 100%, recall 95% for German-German ($\omega = 0.5$)